10-30-00

A

"Express Mail" Mailing Label No. EL 521 771 105 US
Date of Deposit October 27, 2000

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above, with sufficient postage affixed, and is addressed to BOX PATENT APPLICATION, Assistant Commissioner for Patents, Washington, D.C. 20231.

Lucy Flemings

(Typed or printed name of person mailing paper or fee)

(Signature of person mailing paper or fee)

Patent
Attorney's Docket No. 032001-074

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

UTILITY PATENT
APPLICATION TRANSMITTAL LETTER

Box PATENT APPLICATION
Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

Enclosed for filing is the utility patent application of Daniel J. Pugh and Mark Rollins for GOLD CODE GENERATOR DESIGN.

Also enclosed are:

[X] 9 sheet(s) of [ ] formal [X] informal drawing(s);

[ ] a claim for foreign priority under 35 U.S.C. §§ 119 and/or 365 is [ ] hereby made to _ filed in _ on _;

[ ] in the declaration;

[ ] a certified copy of the priority document;

[ ] a General Authorization for Petitions for Extensions of Time and Payment of Fees;

[X] an Assignment document;

[X] an Information Disclosure Statement; and

[X] Other: Appendix.

[X] A [ ] executed [X] partially executed declaration of the inventor(s)

[X] also is enclosed [ ] will follow.

[ ] Please amend the specification by inserting before the first line the sentence --This application claims priority under 35 U.S.C. §§ 119 and/or 365 to _ filed in _ on _; the entire content of which is hereby incorporated by reference.--

[X] Small entity status is hereby claimed.

21839

(10/00)

[X] The filing fee has been calculated as follows [ ] and in accordance with the enclosed preliminary amendment:

CLAIMS					
	NO. OF CLAIMS		EXTRA CLAIMS	RATE	FEE
Basic Application Fee					$710.00 (101)
Total Claims	22	MINUS 20 =	2	× $18.00 (103) =	36.00
Independent Claims	4	MINUS 3 =	1	× $80.00 (102) =	80.00
If multiple dependent claims are presented, add $270.00 (104)					
Total Application Fee					826.00
If small entity status is claimed, subtract 50% of Total Application Fee					413.00
Add Assignment Recording Fee $ if Assignment document is enclosed					40.00
TOTAL APPLICATION FEE DUE					**453.00**

[ ] This application is being filed without a filing fee. Issuance of a Notice to File Missing Parts of Application is respectfully requested.

[X] A check in the amount of $ 453.00 is enclosed for the fee due.

[ ] Charge $ _____ to Deposit Account No. 02-4800 for the fee due.

[X] The Commissioner is hereby authorized to charge any appropriate fees under 37 C.F.R. §§ 1.16, 1.17 and 1.21 that may be required by this paper, and to credit any overpayment, to Deposit Account No. 02-4800. This paper is submitted in duplicate.

Please address all correspondence concerning the present application to:

Robert E. Krebs
BURNS, DOANE, SWECKER & MATHIS, L.L.P.
P.O. Box 1404
Alexandria, Virginia 22313-1404.

Respectfully submitted,

BURNS, DOANE, SWECKER & MATHIS, L.L.P.

Date: October 27, 2000

By: _____
Joseph P. O'Malley
Registration No. 36,226

P.O. Box 1404
Alexandria, Virginia 22313-1404
(650)622-2300

"Express Mail" Mailing Label No. EL 521 771 105 US
Date of Deposit October 27, 2000

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above, with sufficient postage affixed, and is addressed to BOX PATENT APPLICATION, Assistant Commissioner for Patents, Washington, D.C. 20231.

Lucy Flemings

(Typed or printed name of person mailing paper or fee)

(Signature of person mailing paper or fee)

PATENT
Atty Dkt. No. 032001-074

GOLD CODE GENERATOR DESIGN

BACKGROUND OF THE PRESENT INVENTION

The present application concerns pseudo-random generators, in particular gold code generators.

Pseudo-random generators have applicability for a number of communication systems, in particular, for spread spectrum wireless communications. In spread spectrum transmissions, the circuit artificially spreads the transmitted signals bandwidth by modulating an information signal, either in phase or frequency, with a pseudo-random sequence that occurs at a greater rate than that required for the data alone. During signal reception, the receiver synchronizes an internal pseudo-random generator to the pseudo-random sequence of the transmitted signal to fully recover the available power and decode the message. Most direct sequence spread spectrum systems pseudo-randomly modulate the phase of the RF carrier signal 10 times or greater than the rate required for the data transmission. This results in a signal spectrum which is much broader than would be occupied if the RF carrier signal were modulated by only the data stream. Frequency hopping systems use the pseudo-random generator to implement frequency hops within the spread spectrum range.

Matching pseudo-random generators at the transmitter and receiver allow the correlation and recovery of the information signal. Other transmitted signals with different pseudo-random codes can be transmitted in the same bandwidth since the correlation between the different pseudo-random codes is quite low. The transmissions using the different pseudo-random codes will tend not to significantly interfere with one another.

One way of implementing a pseudo-random generator is with a linear feedback shift register (LFSR). Taps from the linear feedback register are sent to a logic circuit to create a new input (feedback) bit. The linear shift register runs through a large number of different codes before repeating. The linear feedback shift register is preferably selected with a feedback path producing the maximum code length. Also beneficial for the linear shift register is low auto-correlation with shifts in the pseudo-random sequence and low cross-correlation with other sequences.

One preferred way of implementing a pseudo-random sequence is to combine the outputs of two linear feedback shift registers. Such a pseudo-random generator is called a gold code generator.

Figure 1 illustrates a gold code generator used with the UMTS European wireless standard. The gold code generator is constructed of two linear feedback shift registers. The first linear feedback shift register has feedback taps from registors 0 and 3. The second linear feedback shift register has feedback taps at registors 0, 1, 2, and 3. The first serial linear feedback shift register's output is combined with the output of the second linear feedback shift register in the EXCLUSIVE-OR 40. The first linear feedback shift register 42 has taps at registers 4, 7 and 18 that go to a EXCLUSIVE-OR (mask) 44. The second linear feedback shift register 46 has taps at registers 4, 6, and 17 fed to the EXCLUSIVE-OR 48. The output to the EXCLUSIVE-ORs 44 and 48 are sent to a second output EXCLUSIVE-OR 52.

It is desired to have an improved implementation of a gold code generator.

SUMMARY OF THE PRESENT INVENTION

The inventors have noticed that the range of taps used to implement the second output of the UMTS gold code generator standard is quite broad: In the first linear shift register between taps 4 and 18 and in the second linear shift register between taps 4 and 17. This broad range of taps makes it difficult to implement a parallel implementation of the gold code generator in arithmetic logic units or other computational units that operate on parallel data.

Since the second output is in fact a delayed version of the first output, the gold code generator can be implemented by forming two pairs of linear feedback shift registers and using different seeds for the second pair of linear feedback shift registers. This significantly reduces the range of the output taps in any of the linear feedback shift registers, and makes it easier to implement a parallel implementation of the gold code generator to produce multiple output bits.

One embodiment of the present invention comprises a gold code generator comprising two pairs of linear feedback shift registers wherein the second seed values for the second pair of linear feedback shift registers are different from the first seed values for the first pair of linear feed back state machines. The second seed values are calculated from the first seed values, wherein the first and second pair of linear feedback shift registers are implemented to produce more than one input bit and more than one output bit for each linear feedback shift registers at the same time.

Another embodiment of the present invention comprises at least one reconfigurable chip implementing a gold code generator, the at least one reconfigurable chip including background and foreground configuration memories. The background configuration memory is adapted such that it can be loaded with a gold code generator configuration while the at least one reconfigurable chip

configured with the foreground configuration operates. After the background configuration is loaded with the gold code generator configuration, the background plane is activated to reconfigure the at least one reconfigurable chip.

Another embodiment of the present invention comprises a method of implementing a pseudo-random code generator comprising the steps of converting a pseudo-random code generator specification into an equivalent representation. The pseudo-random code generator specification being such that taps used to calculate an output include at least one tap within *n* spaces of the input. Equivalent representation is such that no taps are within *n* spaces from the input. The method includes implementing the equivalent representation such that *n* new state bits are calculated at the same time.

Another embodiment of the present invention is a method of implementing a pseudo-random code generator, the method comprising converting a pseudo-random code generation specification into an equivalent of representation. The pseudo-random code generator specification being such that taps used to calculate an output bit are defined within a first shift register span, the equivalent representation is such that taps used to calculate an output bit within a smaller shift register span. The method includes implementing the equivalent representation such that multiple new bit states are calculated at the same time.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a diagram of a prior art gold code generator.

Figure 2 shows the diagram illustrating the implementation of the gold code generator of Figure 1 into an equivalent representation.

Figure 3 is a diagram of that illustrates a parallel implementation of the gold code generator.

Figure 4 is a diagram of a functional block diagram of a gold code generator.

Figure 5 is a diagram of a reconfigurable chip which can be used for implementing the gold code generator of the present invention.

Figure 6A and Figure 6B show a method of switching in the gold code configuration used with one embodiment of the present invention into a reconfigurable fabric of a reconfigurable chip.

Figure 7 is a diagram illustrating the Galois field calculations for the seed values.

Figure 8A and Figure 8B are tables illustrating the values of the lookup tables of Figure 3.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Figure 2 illustrates the conversion of the gold code generator 60 as defined in the UMTS specification into an equivalent representation using two pairs of linear feedback shift registers, pairs 62 and 64. The output from unit 66 of the gold code generator 60 is equivalent to a delayed sequence of the output unit 68. In the present invention, multiple pairs of the linear feedback shift register are used, the second pair of linear feedback shift registers uses a second pair of seed values such that output of the second pair of linear feedback shift registers 64 is a delayed sequence of the sequence produced by the output 66 of the gold code generator 60.

Although the equivalent representation uses more resources, this equivalent representation can be implemented in a parallel implementation that produces multiple output bits. Such a representation is especially useful when implemented with a reconfigurable chip in which reconfigurable elements are configured by configuration bits. One reason why it is easier to do a parallel implementation of the equivalent representation is that a narrow range of output taps is used with the equivalent representation. The output 66 of the gold code generator 60 is quite broad. This makes it difficult to calculate multiple output bits with the standard

gold code representation. In the equivalent representation, the linear feedback shift register pair 64 has only two output bits going to the EXCLUSIVE-OR 70. Additionally, the taps used to produce the feedback states remain relatively close together for both of the pairs of the linear feedback shift registers.

The second shift register 64 is seeded with a new initial seed. This new initial seed value can be calculated before operation of the gold code generator. This calculation is a Galois field calculation which can be done as shown in Figure 7. Additional discussion of the Galois field calculation is given in Chapter 6 entitled "Theory and application of pseudo-random sequences" in the reference "CDMA Systems Engineering Handbook" by Lee and Miller, which is incorporated herein by reference.

Additionally, a copy of the C code to calculate the seed values including some corrections to the math of the Lee and Miller book are enclosed as Appendix II to this application.

In a preferred embodiment, each of the linear feedback shift registers is reset to an initial value at the beginning of each frame. The seed for the linear feedback shift register LFSRA is assigned by the network controller for each user. The seed for the linear feedback shift register LFSRB is 0 x 1FFFFF for all users. The seed for the linear feedback shift register LFSRC is the contents of the linear feedback shift register LFSRA seed shifted by 16,777,232,000 cycles, and can be computed by a processor at the beginning of each call. The seed for linear feedback shift register LFSRD is 0 x 1FFFFF shifted by 16,777,232,000 cycles for each user and thus is a static constant value which can be precalculated and stored. The seed for all of the linear feedback shift registers can be stored in memory elements in a reconfigurable fabric.

Figure 3 shows a parallel implementation of one of the linear feedback shift registers of the equivalent representation. The span of taps to produce an output bit is quite small so no complicated logic is required to calculate an output. The

input bits are also calculated in parallel using lookup tables. Note how a parallel implementation spreads the number of taps needed for the feedback or output calculations. There is no requirement for a really wide span lookup table for the output bits since the equivalent representation is used.

Figures 8A and 8B show the table lookup values of one embodiment of the lookup tables of Figure 3.

Figure 4 illustrates a functional block diagram of a gold code generator in one embodiment. The gold code generator of one embodiment deals with multiple users, each user having a different input seed thus producing different sequence of the gold code generator, each of the different sequences having a relatively low cross-correlation. In this embodiment, for each seed the delayed version for the second pair of shift registers must be calculated using the Galois field calculation described above.

Figure 5 illustrates a reconfigurable chip 80. The reconfigurable chip 80 implements the gold code generator in one embodiment. The reconfigurable chip 80 includes a reconfigurable fabric 82 which can be configured into a variety of configurations. The CPU 88 can be used for the LFSRC seed calculations that are difficult to do in the reconfigurable fabric.

In a preferred embodiment, the reconfigurable chip 80 includes the processor 88 such as a reduced instructions set computing (RISC) central processing unit (CPU). In one embodiment the CPU 88 runs portions of the algorithms which are difficult to implement in the reconfigurable fabric.

In one embodiment, the reconfigurable fabric is configured by a foreground configuration plane 84. While the foreground configuration plane 84 is operating, the background configuration can be loaded from the background plane 86. The reconfigurable fabric 82 in a preferred embodiment includes a number of configurable data path units, memory elements, and interconnect elements.

In one embodiment, the data path units include comparators, an arithmatic unit (ALU), and registers which are configurable to implement operations of the algorithm. In one embodiment the reconfigurable fabric 82 also includes dedicated elements such as multiple elements and memory elements. The memory elements can be used for storing algorithm data.

Figures 6A and 6B illustrate the operation of the reconfigurable fabric to the system of the present invention. In Figure 6A, the gold code generator configuration is loaded into the background configuration plane 90. The foreground configuration plane 92 is configured with another configuration so as to configure the reconfigurable fabric with that configuration. In Figure 6B the configuration plane in background plane is loaded into the foreground plane. This almost instantaneously configures the reconfigurable fabric 92' into the gold code generator configuration.

Details of the implementation of the gold code generator are shown in Appendix I, especially on pages 42-64.

Although only preferred embodiments of the invention are specifically disclosed and described above, it will be appreciated that many modifications and variations of the present invention are possible in light of the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention.

Claims:

1. A gold code generator comprising:

two pairs of linear feedback shift registers, wherein second seed values for the second pair of linear feedback shift registers are different from first seed values for the first pair of linear feedback shift registers, the second seed values being calculated from the first seed values, wherein the first and second pair of linear feedback shift registers are implemented to produce more than one new state bit and more than one output bit for each linear feedback shift registers at the same time.

2. The gold code generator of claim 1 wherein the seed values for the second pair of linear feedback shift registers are delayed values of the first seed values.

3. The gold code generator of claim 1 wherein the gold code generator is implemented on a reconfigurable logic chip.

4. The gold code generator of claim 3 wherein the calculation of some of the second seed values is done using a dedicated processor on the reconfigurable chip.

5. The gold code generator of claim 3 wherein the gold code generator configuration is loaded into a background plane of the reconfigurable chip, while the reconfigurable chip is operating on another configuration in the foreground.

6. The gold code generator of claim 3 wherein the feedback is implemented using lookup tables.

7. A system comprising:

at least one reconfigurable chip implementing a gold code generator, the at least one reconfigurable chip including background and foreground configuration memories, wherein the background configuration memory is adapted such that it can be loaded with a gold code generator configuration while the at least one reconfigurable chip, configured with the foreground plane, operates, and wherein after the background configuration loads, the gold code generator configuration can be activated to reconfigure the at least one reconfigurable chip.

8. The system of claim 7 wherein the gold code generator comprises two pairs of linear feedback shift registers wherein the seed values for the second pair of linear feedback shift registers is different from the seed values for the first pair of linear feedback shift registers.

9. The gold code generator of claim 7 wherein the second seed values are calculated from the first seed values.

10. The gold code generator of claim 9 wherein the calculation of the second seed values is done at least partially in a processor on the reconfigurable chip.

11. The system of claim 10 in which the gold code generator is implemented to produce more than one output bit at the same time.

12. A method of implementing a pseudo-random code generator:

converting a pseudo-random code generator specification into an equivalent representation, the pseudo-random code generator specification being such that taps used to calculate an output include at least one tap within n spaces from the

input, the equivalent representation is such that no such taps are within n spaces from the input; and

implementing the equivalent representation such that multiple new state bits are calculated at the same time.

13. The method of claim 12 wherein the pseudo-random code generator specification being such that taps to calculate an output is defined within a first chip register span, the equivalent representation is such that taps to calculate an output bit are within a smaller shift register span.

14. The method of claim 12 wherein the equivalent representation includes two pairs of linear feedback shift registers wherein the second seed values for the second pair of linear feedback shift registers is different from a first seed value for the first pair of linear feedback shift registers.

15. The method of claim 12 wherein the pseudo-random code generator comprises a gold code generator.

16. The method of claim 12 wherein the pseudo-random code generator is implemented on a reconfigurable chip.

17. A method of implementing a pseudo-random code generator comprising:

converting a pseudo-random code generator specification into an equivalent representation, the pseudo-random code generator specification being such that taps to calculate an output are defined within a first shift register span, the equivalent representation is such that the taps to calculate an output bit are within a smaller shift register span; and

implementing the equivalent representation such that multiple output bits are calculated at the same time.

18. The method of claim 17 wherein the pseudo-random code generation specification is such that taps used to calculate an output have at least one tap within n spaces from the input, the equivalent representation is such that no such tap is within n spaces from the input.

19. The method of claim 17 wherein two pairs of linear feedback shift registers are used in the equivalent representation.

20. The method of claim 19 wherein the second seed values for the second pair of linear feedback shift registers are different from the first seed values for the first pair of linear feedback shift registers.

21. The method of claim 17 implemented on a reconfigurable chip.

22. The method of claim 17 wherein in the equivalent representation of the output bits are calculated from taps at a single register for each linear feedback shift register.

Abstract

A gold code generator is described comprising two pairs of linear feedback shift registers, the seed values for the second pair of linear feedback shift registers are different from the seed values for the first pair of linear feedback shift registers. The second seed values are calculated from the first seed values. The use of this second pair of linear feedback shift registers prevents the need to use a wide span of taps to the linear feedback shift register to produce output bits. By using two pairs of linear feedback shift registers, a parallel output implementation can be produced in which multiple output bits are produced in a single clock cycle.

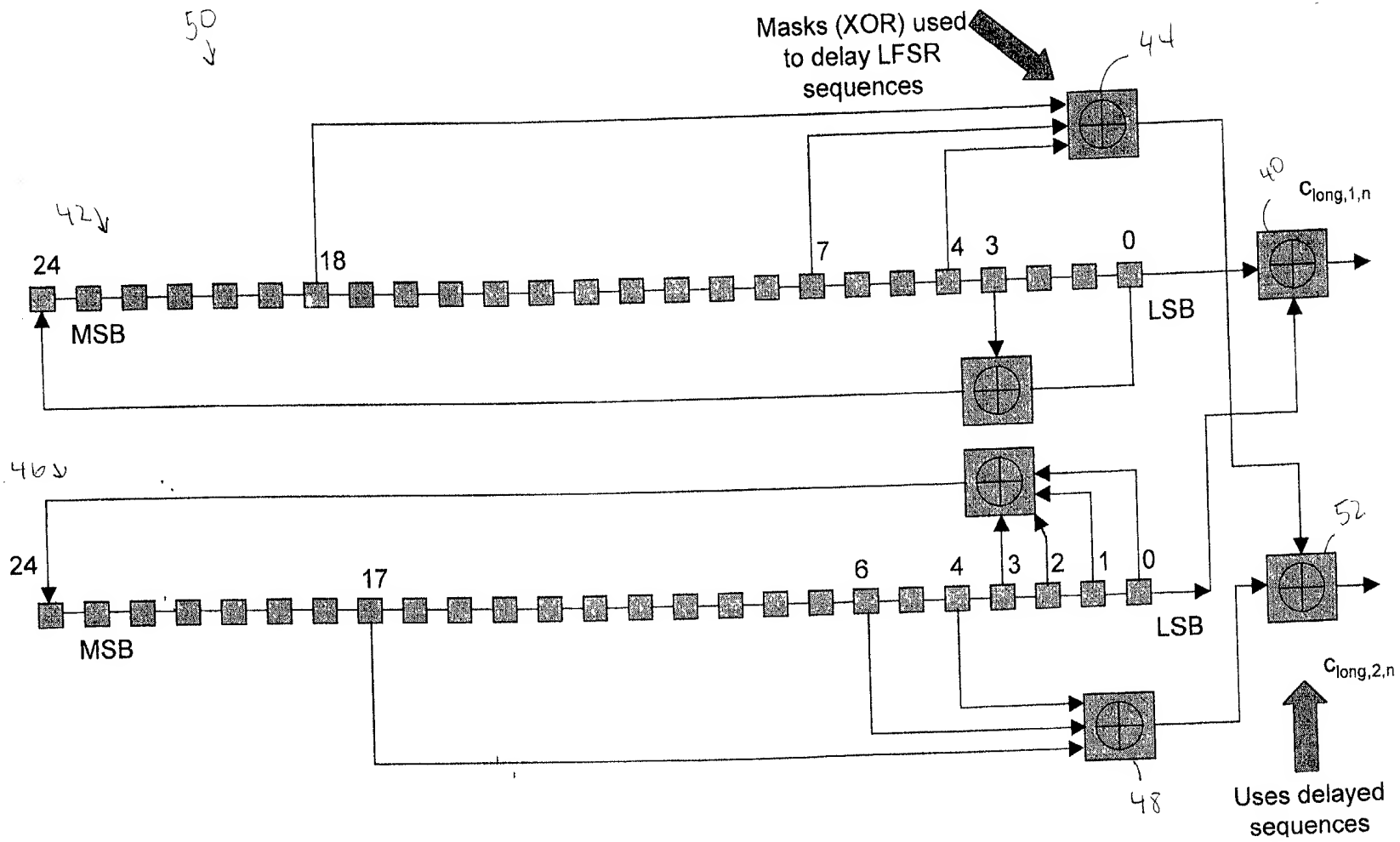


FIGURE 1
(PRIOR ART)

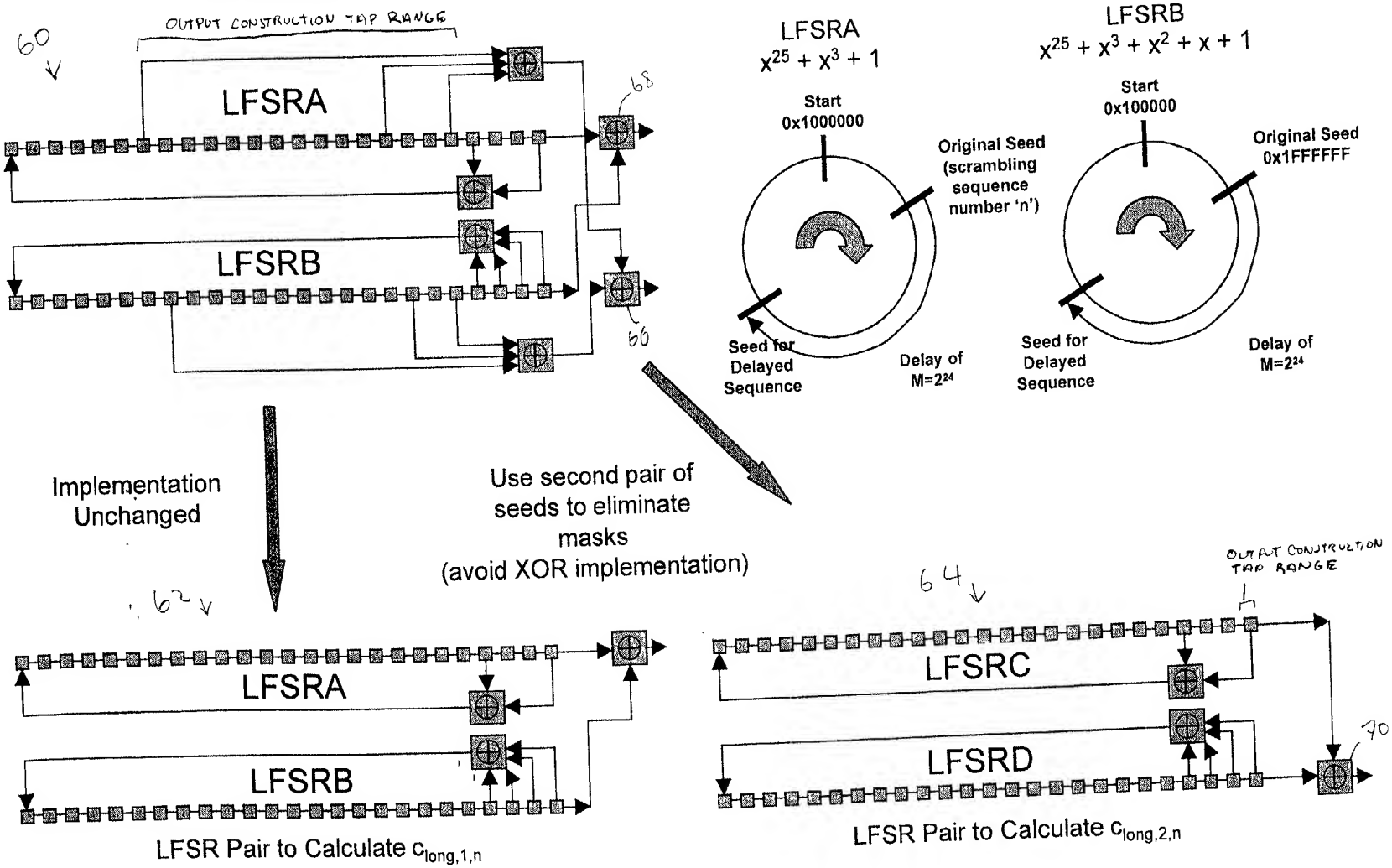

60
OUTPUT CONSTRUCTION TAP RANGE
LFSRA
LFSRB
68
66
LFSRA
$x^{25} + x^3 + 1$
Start
0x1000000
Original Seed
(scrambling
sequence
number 'n')
Seed for
Delayed
Sequence
Delay of
$M=2^{24}$
LFSRB
$x^{25} + x^3 + x^2 + x + 1$
Start
0x100000
Original Seed
0x1FFFFFF
Seed for
Delayed
Sequence
Delay of
$M=2^{24}$
Implementation
Unchanged
Use second pair of
seeds to eliminate
masks
(avoid XOR implementation)
62
LFSRA
LFSRB
LFSR Pair to Calculate $c_{long,1,n}$
64
OUTPUT CONSTRUCTION
TAP RANGE
LFSRC
LFSRD
70
LFSR Pair to Calculate $c_{long,2,n}$

FIGURE 2

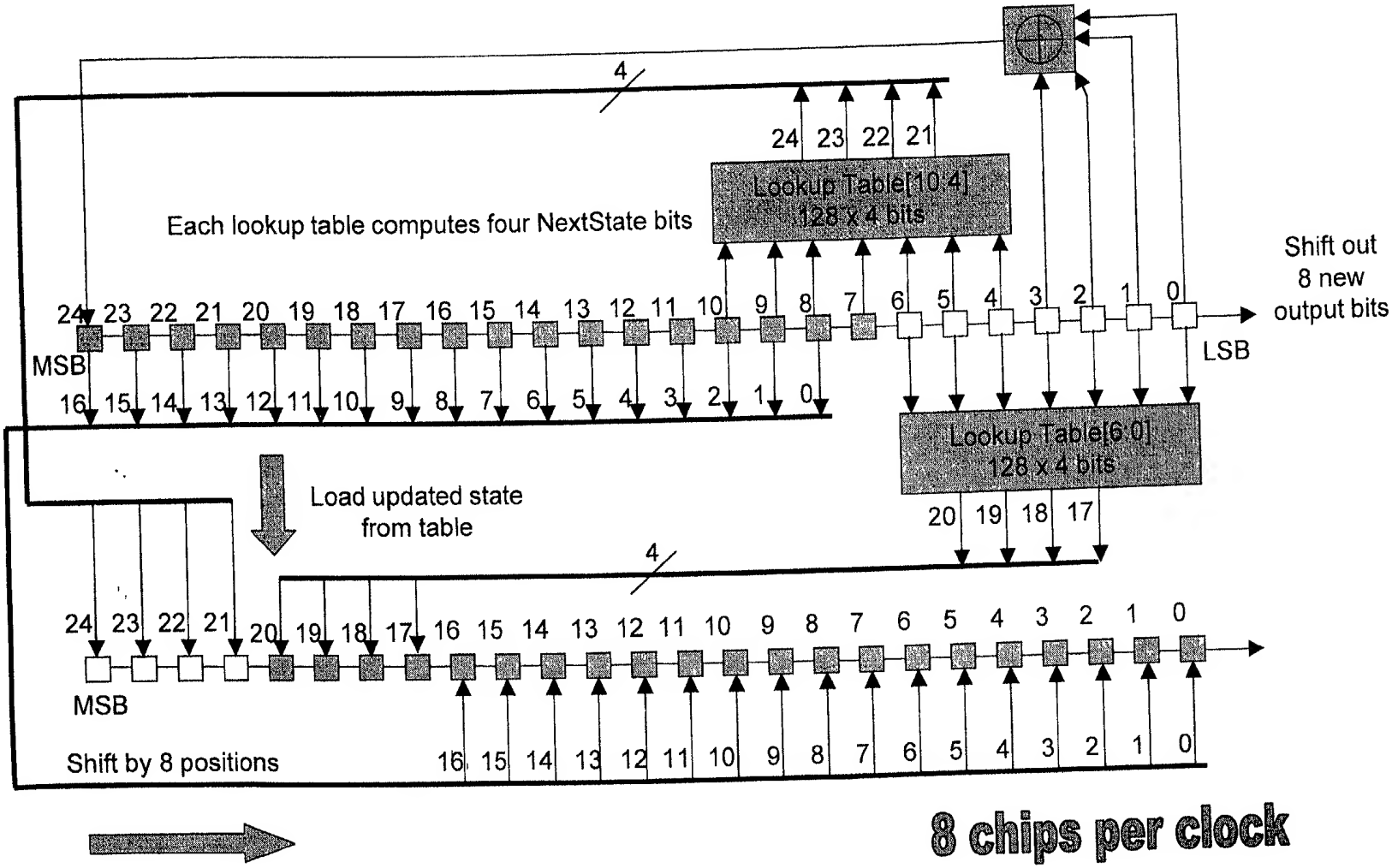


FIGURE 3

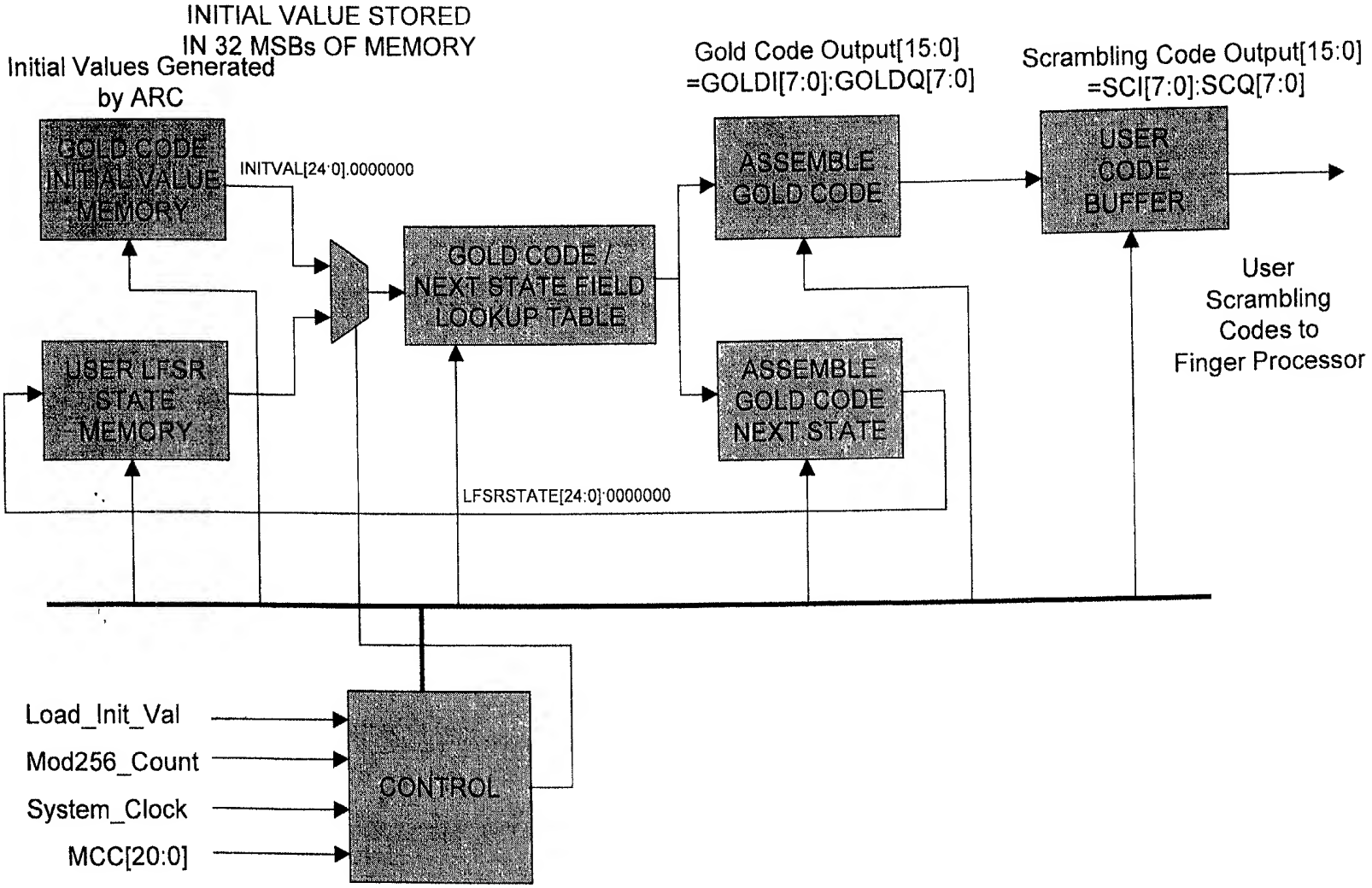

INITIAL VALUE STORED
IN 32 MSBs OF MEMORY
Initial Values Generated
by ARC
GOLD CODE
INITIAL VALUE
MEMORY
INITVAL[24:0].0000000
GOLD CODE /
NEXT STATE FIELD
LOOKUP TABLE
USER LFSR
STATE
MEMORY
LFSRSTATE[24:0]:0000000
Gold Code Output[15:0]
=GOLDI[7:0]:GOLDQ[7:0]
ASSEMBLE
GOLD CODE
ASSEMBLE
GOLD CODE
NEXT STATE
Scrambling Code Output[15:0]
=SCI[7:0]:SCQ[7:0]
USER
CODE
BUFFER
User
Scrambling
Codes to
Finger Processor
Load_Init_Val
Mod256_Count
System_Clock
MCC[20:0]
CONTROL

FIGURE 4

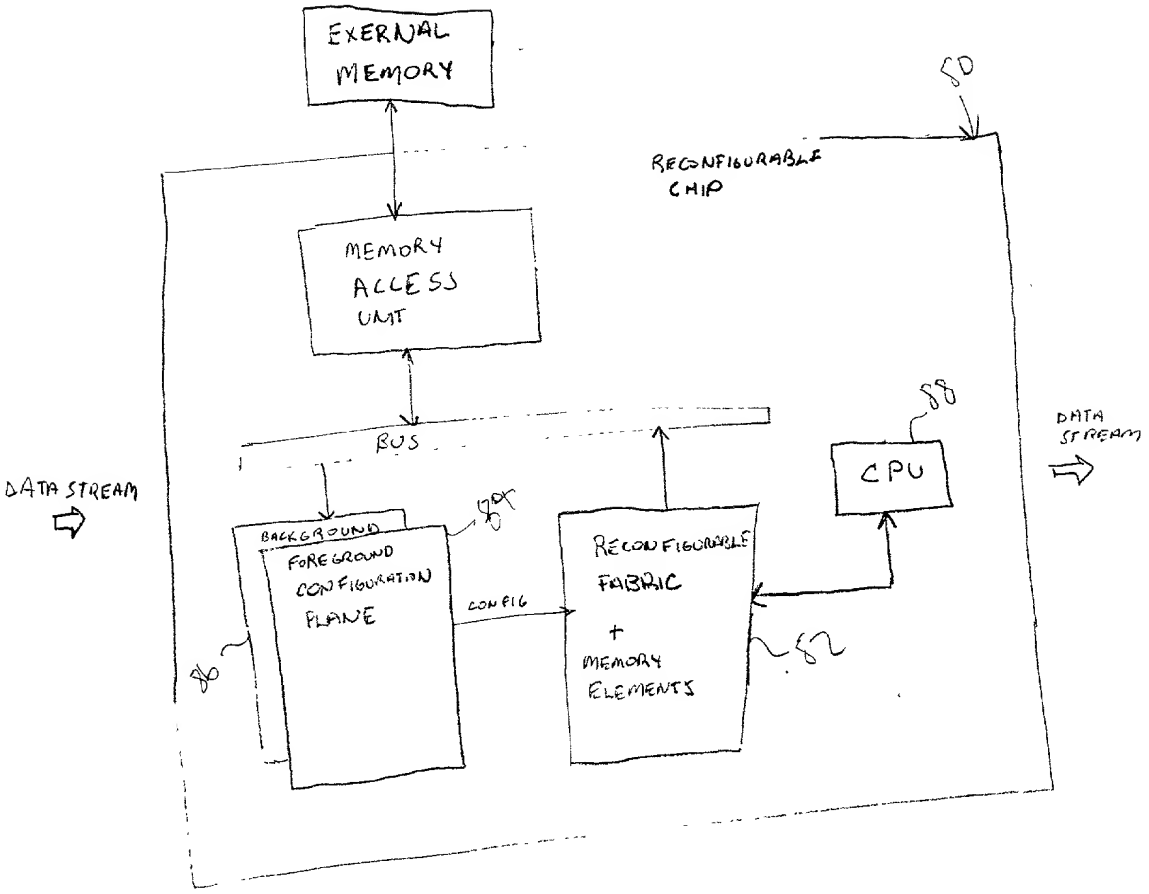

EXERNAL MEMORY
50
RECONFIGURABLE CHIP
MEMORY ACCESS UNIT
BUS
88
CPU
DATA STREAM
DATA STREAM
BACKGROUND
FOREGROUND CONFIGURATION PLANE
CONFIG
RECONFIGURABLE FABRIC
+
MEMORY ELEMENTS
86
82

FIGURE 5

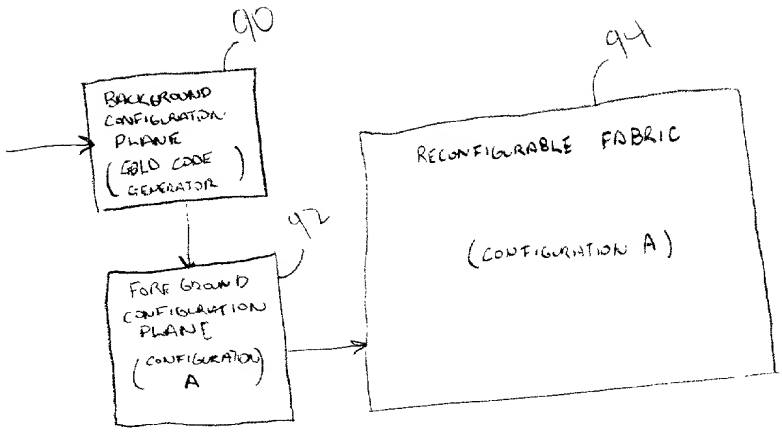


FIGURE 6A

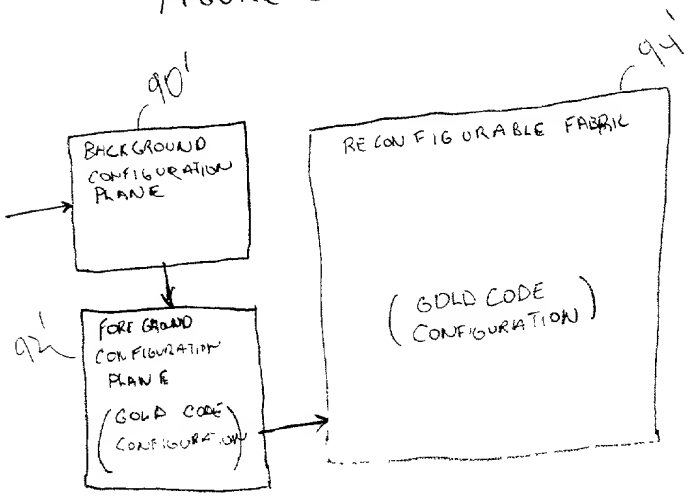


FIGURE 6B

$C_{long1,n}$ = LSFRA[7:0] XOR LSFRB[7:0]
Let us define LFSRC'[i] = LSFRC[2⌊i/2⌋]

$C_{long,n}(i) = C_{long1,n}(i)(1 + j(-1)^i(c_{long2},n(2\lfloor i/2 \rfloor))$ (from 3G TS25.213)

Multiplying bits by +1/-1 is the same as XOR for 0s and 1s.
XORing by 0xAA can be used in place of the $(-1)^i$ term.
In binary representation, the Scrambling Code $C_{long,n}$ becomes:

$C_{long,n}[7:0] = C_{long1,n}[7:0](1 + j(0xAA) \text{ XOR } C'_{long2,n}[7:0])$

$C_{long,n}[7:0]$ = LFSRA[7:0] XOR LFSRB[7:0]
+j(LFSRA[7:0] XOR LFSRB[7:0] XOR 0xAA XOR LFSRC'[7:0] XOR LFSRD'[7:0]

$C_{long,n}[7:0]$ = SCI[7:0] + jSCQ[7:0]

Let us define LFSRD''[7:0] = 0xAA XOR LFSRD'[7:0], then:

$C_{long,n}[7:0]$ = (LFSRA[7:0] XOR LFSRB[7:0])
+ j(LFSRA[7:0] XOR LFSRB[7:0] XOR LFSRC'[7:0] XOR LFSRD''[7:0])

We use a lookup table to compute LFSRC'[7:0] and LFSRD''[7:0])

FIGURE 7

Gold Code Generator Lookup[6:0] Definitions

At Address 4n+0: OUT[7:0] = Next StateA[3:0]:PASSA[3:0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]	At Address 4n+2: OUT[7:0] = Next StateC[3:0]:LFSRC'[3:0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0]
At Address 4n+1: OUT[7:0] = Next StateB[3:0]:PASSB[3:0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]	At Address 4n+3: OUT[7:0] = Next StateD[3:0]:LFSRD''[3:0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = /IN[2] OUT[2] = IN[2] OUT[1] = /IN[0] OUT[0] = IN[0]

FIGURE 8A

Gold Code Generator Lookup[10:4] Definitions

At Address 4n+0: OUT[7:0] = IN[7:4]:Next StateA[7:4] OUT[7] = IN[3] OUT[6] = IN[2] OUT[5] = IN[1] OUT[4] = IN[0] OUT[3] = IN[6] XOR IN[3] OUT[2] = IN[5] XOR IN[2] OUT[1] = IN[4] XOR IN[1] OUT[0] = IN[3] XOR IN[0]	At Address 4n+2: OUT[7:0] = IN'[7:4]:Next StateC[7:4] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0]
At Address 4n+1: OUT[7:0] = IN[7:4]:Next StateB[7:4] OUT[7] = IN[3] OUT[6] = IN[2] OUT[5] = IN[1] OUT[4] = IN[0] OUT[3] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[2] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[1] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[0] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0]	At Address 4n+3: OUT[7:0] = IN''[7:4]:Next StateD[7:4] OUT[3] = /IN[2] OUT[2] = IN[2] OUT[1] = /IN[0] OUT[0] = IN[0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0]

FIGURE 8B

Chameleon Systems UMTS Rake Receiver Mapping Analysis

27-April-2000

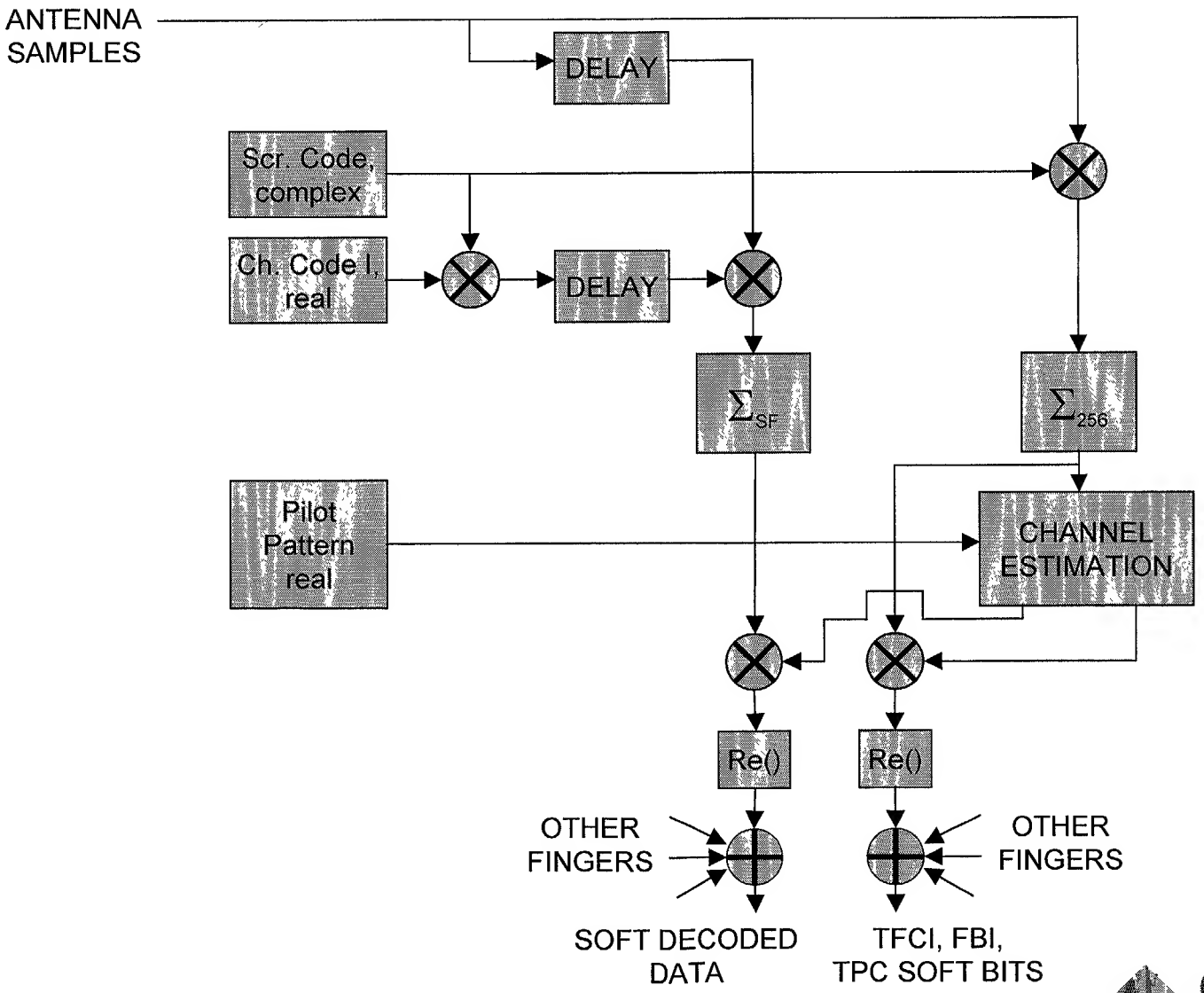
Revision 0.41

Dan Pugh & Mark Rollins
Chameleon Systems, Inc.
dan@chameleonsystems.com
rollins@chameleonsystems.com

Rake Receiver Requirements and Assumptions

- Requirements
  - Implement 32 users Pilot Processing and Data Despreading
    - Strive for target of 32 users in CS2112 (125 MHz)
  - Exceed target of 75 users in CS2112X (250 MHz)
- Assumptions
  - Maximum of 12 antennas
  - Maximum of 8 fingers per user
  - Average of 4 fingers per user
  - Spreading factors of 4 to 256 on DPDCH
  - Dual-port RAM at the input; Input order may be specified
  - ARC supplies scrambling code seeds
  - The Chameleon Processor is running at exactly 32 x the chip rate
  - An external Phase-locked-loop generates the 32 x (122.88) processor clock and is locked to the 3.84 MHz chip clock

Rake Processor Functional Block Diagram

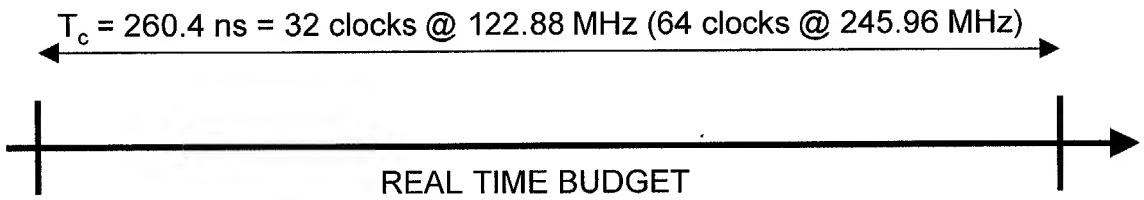


Rake Receiver Description of Pipelined Operation

- Store a window of 128 chip samples sampled $T_c/2$ apart
- In each grouping of 256 consecutive 122.88 MHz clocks:

  At EVERY 122.88 MHz clock period:
  - Read 8 consecutive chip samples at the correct multi-path delay offset
  - Align the eight samples to the Despreading Code (Gold Code)
  - Multiply the 8 data samples by the 8 appropriate Despreading Codes
  - Sum the eight despread chips into one sample (two samples if SF=4)
  - Accumulate the sum-of-eight chips into a single despread symbol
- Send the despread Pilot Symbols to the ARC processor
  - Calculate Channel Estimation Weights
  - Multiply TFCI, FBI, TPC bits by Channel Estimation Weights
  - Sum up to six fingers to form each soft bit (symbol)

Fundamental Scheduling Analysis

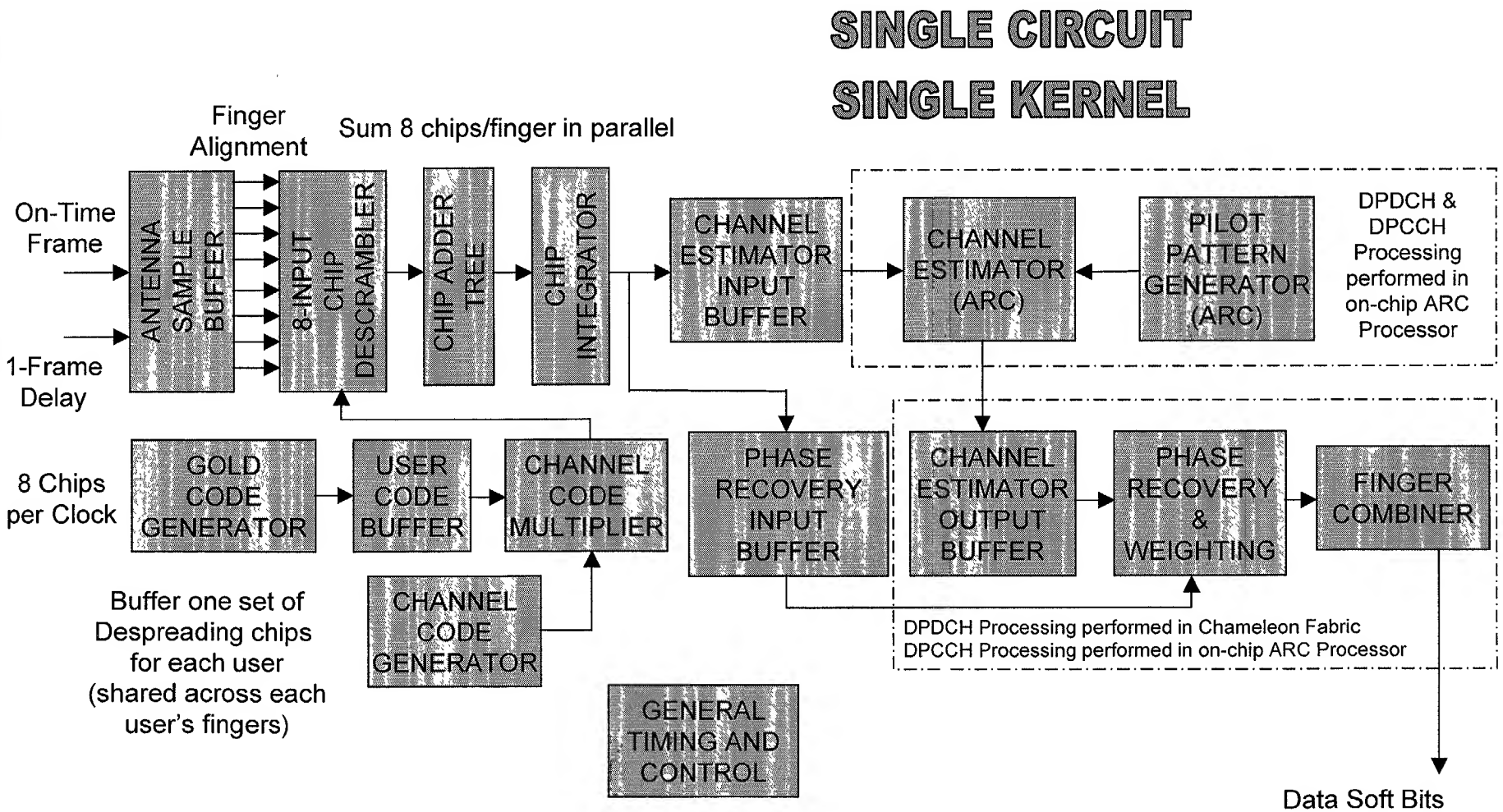


Parallel Implementation:

- Each circuit processes 8 chips per clock
- Throughput of 256 fingers per circuit
- Populate device with single circuit
- Achieves 128 pilot data fingers (32 users) @ 122.88 MHz
- Achieves 128 data fingers (same 32 users) @ 122.88 MHz
- Achieves 512 fingers (64 users) @ 245.96 MHz
- Utilizes single centralized control unit
- Fits within a single Chameleon device

Parallel Implementation is more efficient than multiple instantiations of individual units

Chameleon Rake Processor High-Level Partitioning Overview



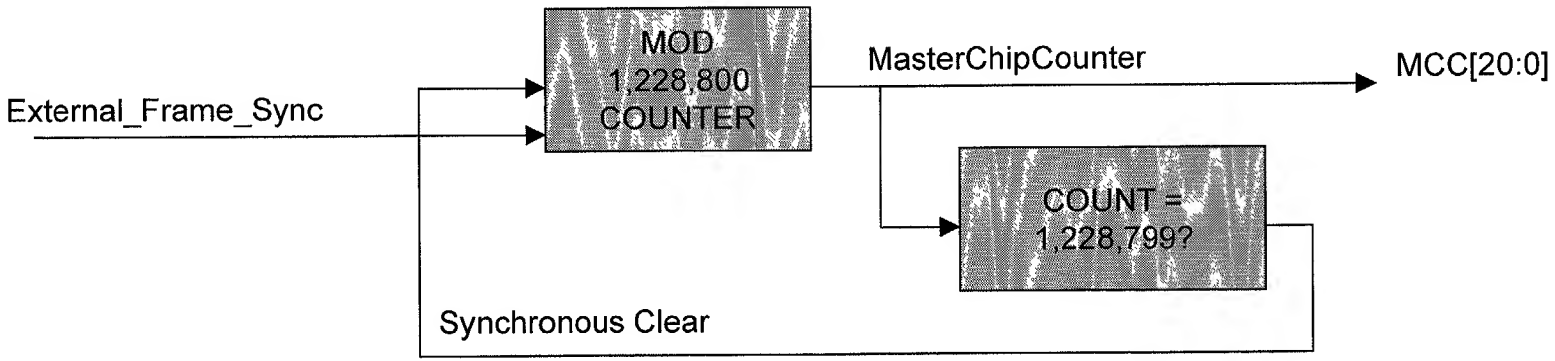
CHAMELEON SYSTEMS, INC.

General Timing and Control

- The Rake Processor is synchronized to the chip-rate clock with an external Phase-Locked-Loop (PLL)
- The Rake Processor clock runs at exactly 32 times the chip-rate clock (32 x 3.84 MHz = 122.88 MHz)
- All processes in the Rake Processor are synchronized to the Master Chip Counter (MCC)
- The MCC counts modulo 1,228,800 (32 clocks/chip x 15 slots x 2560 chips/slot) and is reset by the external Frame Sync signal
- Various bits in the MCC are used to generate the Antenna Sample Buffer Write Address

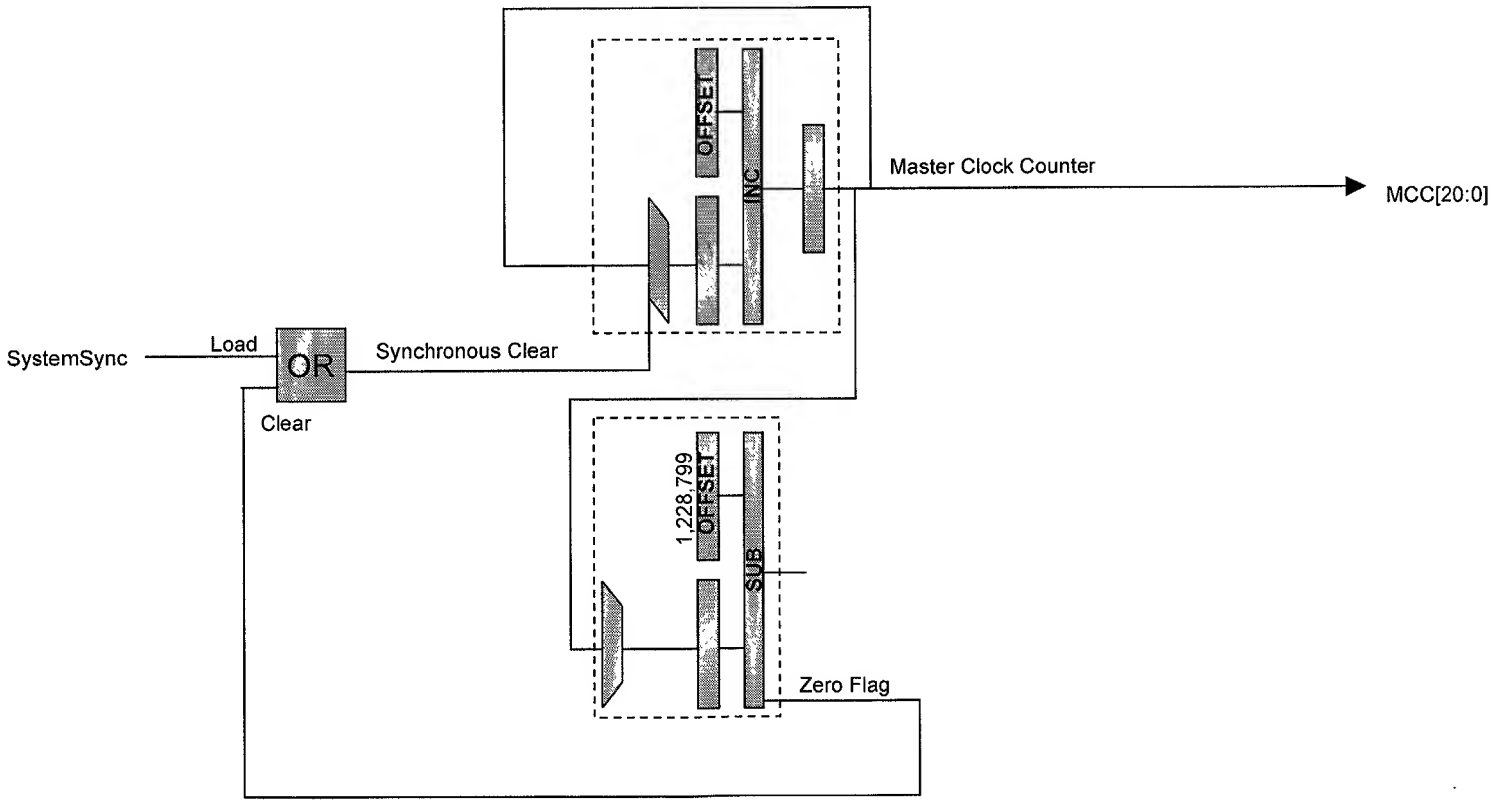
Master Chip Counter Block Diagram

The External_Frame_Sync signal clears the modulo 1,228,800 counter



CHAMELEON SYSTEMS, INC.

Master Chip Counter Implementation



CHAMELEON SYSTEMS, INC.

Master Chip Counter Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - 2 DPUs
  - 0 LSMs

Antenna Sample Buffer Requirements and Assumptions

- Requirements
  - Provide memory in the Antenna Buffer so that a128-chip (256 half-chips) range of samples can be stored for all 12 antennas
  - This allows the Finger Alignment Buffer and the Antenna Buffer to be merged into a single memory
- Assumptions
  - The samples must be organized such that any eight samples T_c apart from one antenna may be read simultaneously from the buffer
  - Each Rake Receiver circuit (Chameleon chip) processor services from one to six sectors
  - Each Sector must process the primary and diversity antennas for the primary sector and the two adjacent sectors, for a minimum of six antennas per sector of coverage
  - Support of all 12 antennas is required

Antenna Sample Buffer General Description

- There are two partitions of the Antenna Sample Buffer
  - The first partition contains the on-time antenna data
  - The second partition contains the on-time antenna data that is delayed by approximately one frame
- The value of the delay is one frame plus the time required to compute the Spreading Factor from the TFCI bits
- The delayed data is required because the TFCI bits are in the same frame as the data that it controls

Antenna Sample Buffer General Description

- The Antenna Sample Buffer is used to store a large enough window of data that all six fingers may be accessed from any of the 12 antennas
- Data for each finger is read from the Antenna Buffer at the offset specified by the Path Searcher
- The Antenna Buffer is organized so that ANY eight consecutive samples T_c apart may be accessed from the buffer in a single clock cycle

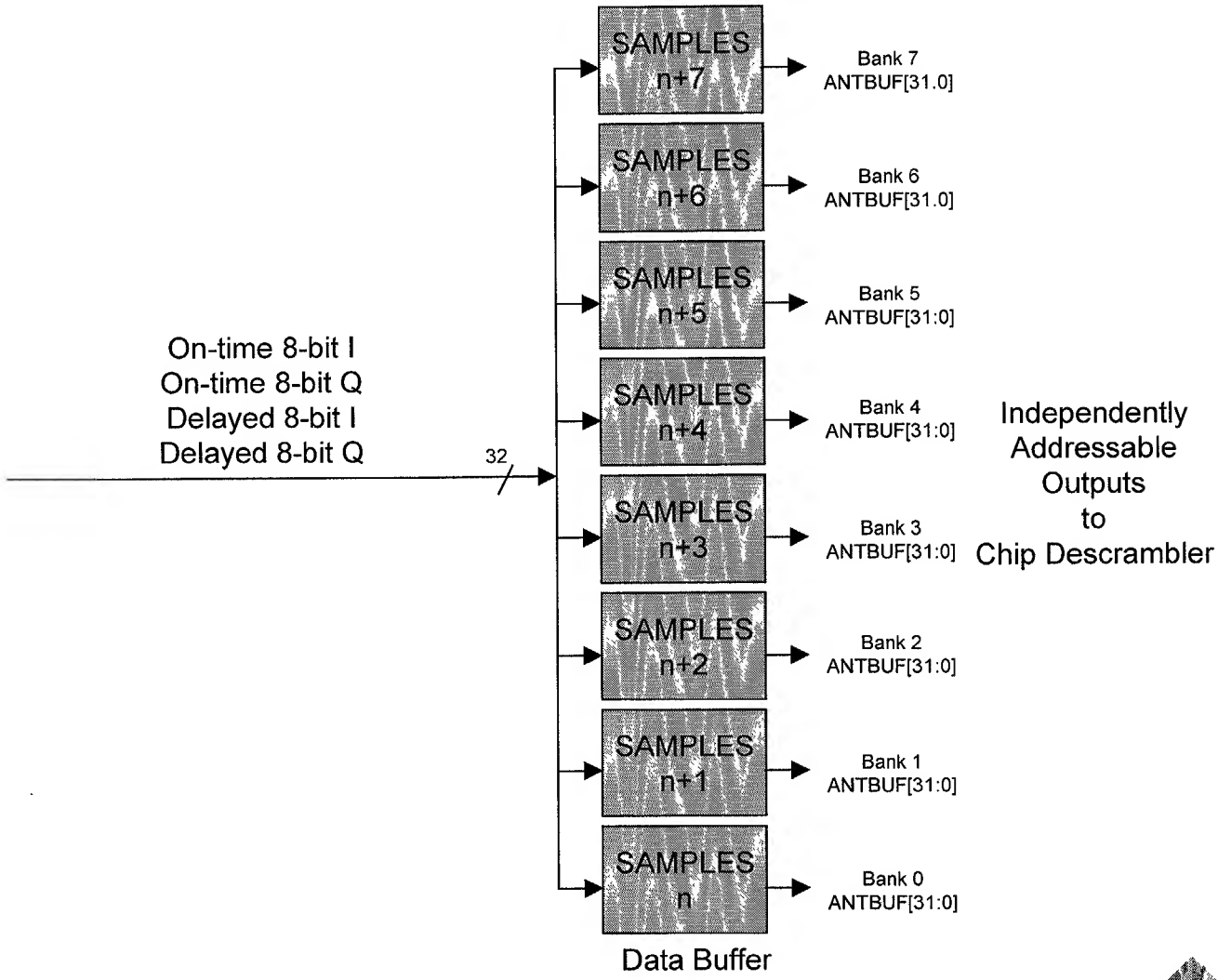
Antenna Sample Buffer Multipath Support Capabilities

- Specifications:
  - Chip-rate = 3.84 MHz (260.4 ns, wavelength = 78.125 m)
  - Maximum distance of mobile user to base station = 7000 m
- The 7000 m user radius corresponds 0-89.6 chips of line-of-sight delay
- The 128-chip Antenna Sample Buffer provides an additional 38.4-chip buffer to support multipath components
- This corresponds to support for multipath components with up to 10,000 ns (3000 m) delay for all 12 antennas
- If the Antenna Sample Buffer is retasked to support only 8 antennas (two adjacent sectors) the maximum multipath support may be increased to 26,664 ns (8000 m)

Antenna Sample Buffer Multipath Support Capabilities

- Given a 5000m user radius (3G TS25.942)
  - A 5000 m user radius corresponds 64 chips of line-of-sight delay
  - The 128-chip Antenna Sample Buffer provides an additional 64-chip buffer to support multipath components
- This corresponds to 10,000m of support for a combination of the user radius plus multipath delay with all 12 antennas
- The Antenna Sample Buffer may be retasked to support:
  - 8 antennas (two sectors) for 15,000m of user radius/multipath support
  - 6 antennas (one sector) for 20,000m of user radius/multipath support

Antenna Sample Buffer Functional Block Diagram



CHAMELEON SYSTEMS, INC.

Antenna Sample Buffer Memory Map for Bank 0

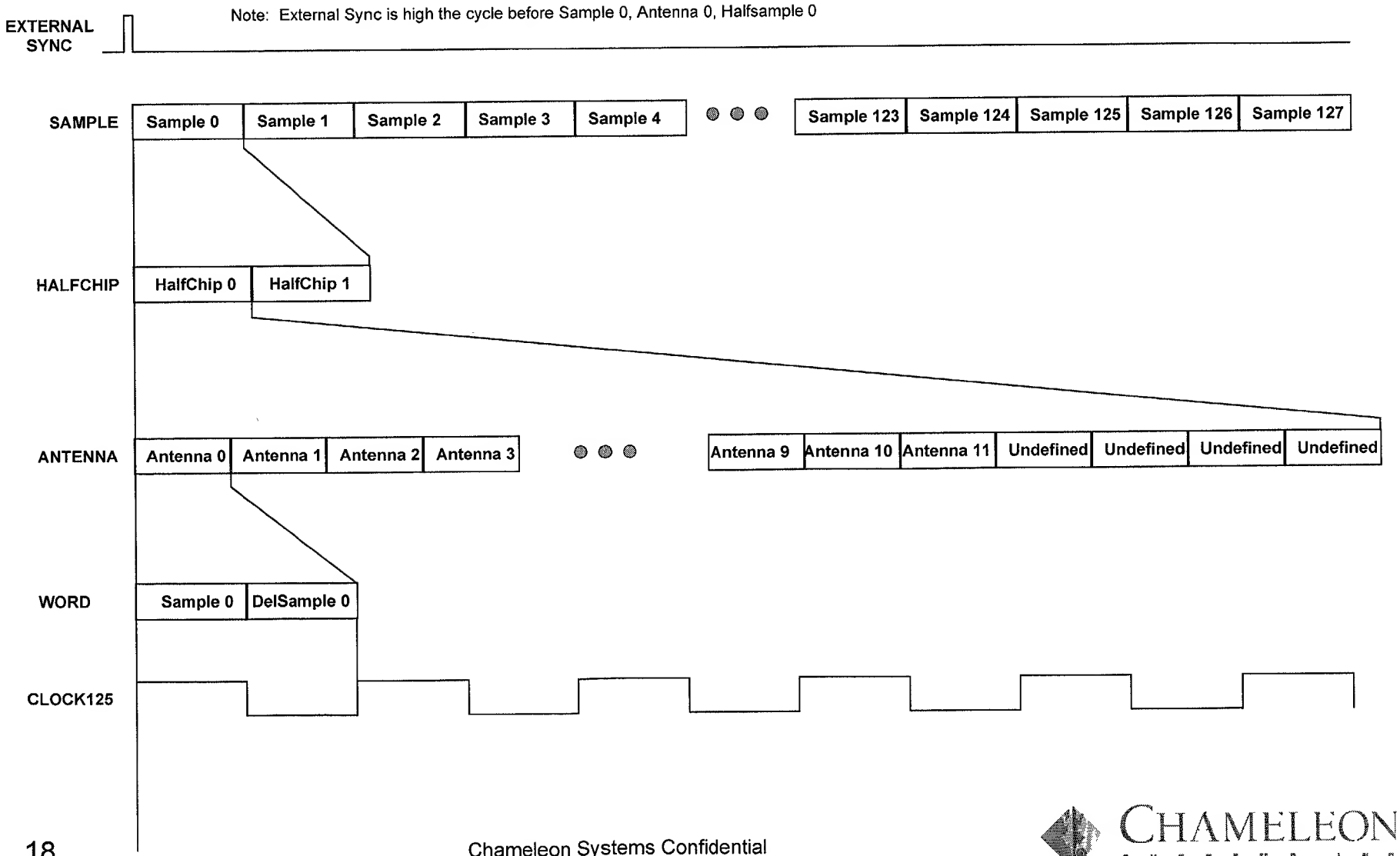
ADDRESS	ANTBUF[31:0]
0x5FC	Antenna 11, Sample 120, HalfChip 1
0x5F8	Antenna 11, Sample 120, HalfChip 0
0x5F4	Antenna 11, Sample 112, HalfChip 1
0x5F0	Antenna 11, Sample 112, HalfChip 0
⋮	⋮
0x094	Antenna 1, Sample 16, HalfChip 1
0x090	Antenna 1, Sample 16, HalfChip 0
0x08C	Antenna 1, Sample 8, HalfChip 1
0x088	Antenna 1, Sample 8, HalfChip 0
0x084	Antenna 1, Sample 0, HalfChip 1
0x080	Antenna 1, Sample 0, HalfChip 0
0x07C	Antenna 0, Sample 120, HalfChip 1
0x078	Antenna 0, Sample 120, HalfChip 0
0x074	Antenna 0, Sample 112, HalfChip 1
0x070	Antenna 0, Sample 112, HalfChip 0
⋮	⋮
0x014	Antenna 0, Sample 16, HalfChip 1
0x010	Antenna 0, Sample 16, HalfChip 0
0x00C	Antenna 0, Sample 8, HalfChip 1
0x008	Antenna 0, Sample 8, HalfChip 0
0x004	Antenna 0, Sample 0, HalfChip 1
0x000	Antenna 0, Sample 0, HalfChip 0

ANTENNA SAMPLE BUFFER WORD CONTENTS

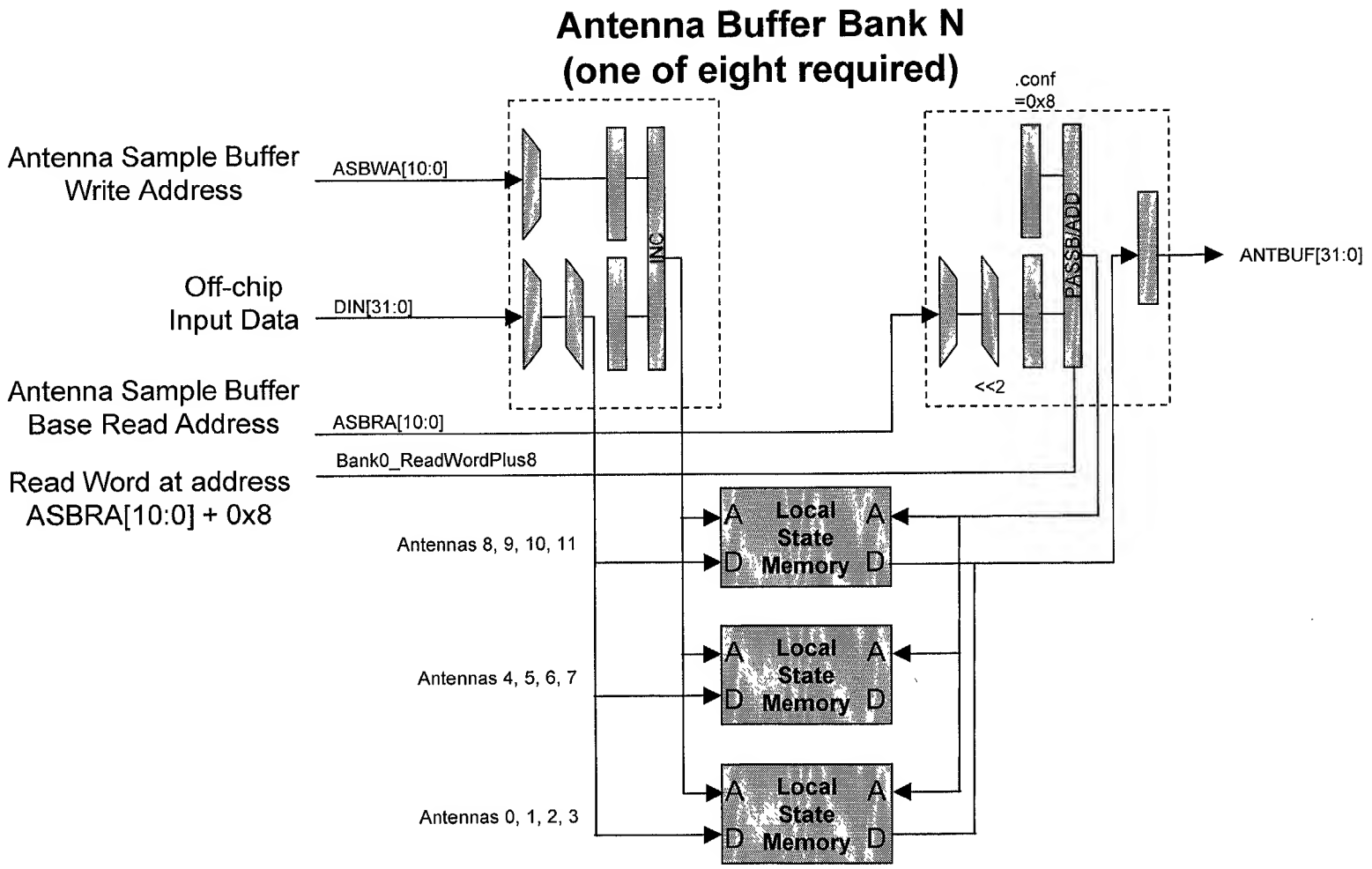
ANTBUF[31:24]	ANTBUF[23:16]	ANTBUF[15:8]	ANTBUF[7:0]
Non-Delayed Q[7:0]	Delayed Q[7:0]	Non-Delayed I[7:0]	Delayed I[7 0]

Note:
Sample n HalfChip 0 is used to denote the sample at time n
Sample n HalfChip 1 is used to denote the sample at time n+1/2

External Antenna Sample Buffer Bus Organization



Antenna Sample Buffer Data Buffer Implementation



Antenna Sample Buffer Write Address Bit Definitions

- The Modulo 1,228,00 Master Chip Counter (MCC) bit fields may be defined with respect to the input data samples to the Antenna Sample Buffer Write Address (ASBWA)
- The Antenna Sample Buffer Write Address Generator bits are MCC bits that have been reordered to properly store the input samples

MCC BIT	DESCRIPTION
MCC[11]	ChipCount[6]
MCC[10]	ChipCount[5]
MCC[9]	ChipCount[4]
MCC[8]	ChipCount[3]
MCC[7]	ChipCount[2]
MCC[6]	ChipCount[1]
MCC[5]	ChipCount[0]
MCC[4]	HalfChip
MCC[3]	Antenna[3]
MCC[2]	Antenna[2]
MCC[1]	Antenna[1]
MCC[0]	Antenna[0]

ANTENNA SAMPLE BUFFER BANK ADDRESS (for each of the eight banks)

ASBWA BIT	DESCRIPTION
ASBWA[10]	Antenna[3]
ASBWA[9]	Antenna[2]
ASBWA[8]	Antenna[1]
ASBWA[7]	Antenna[0]
ASBWA[6]	ChipCount[6]
ASBWA[5]	ChipCount[5]
ASBWA[4]	ChipCount[4]
ASBWA[3]	ChipCount[3]
ASBWA[2]	HalfChip
ASBWA[1]	0
ASBWA[0]	0

Note:
ChipCount[2:0] is used to select one of eight physical memory banks

The one-frame-delayed Samples are stored in bytes 0 and 2 of the Antenna Sample Buffer, while the non-delayed samples are stored in bytes 1 and 3 of the Antenna Sample Buffer

Each Bank requires 3 LSMs

Antenna Sample Buffer Write Address Generation

- If we look at the fields of the Write Address Generator we see that:
  - CASE A: When MCC[11:0]=0xFFF
    - ASBWA[10:7] = Antenna[3:0] must be cleared
    - ASBWA[6:3] = ChipCount[6:3] must be cleared
    - ASBWA[2] = HalfChip must be cleared
  - CASE B: When MCC[7:0]=0xFF
    - ASBWA[10:7] = Antenna[3:0] must be cleared (by incrementing by one)
    - ASBWA[6:3] = ChipCount[6:3] increments by one
    - ASBWA[2] = HalfChip must be cleared
  - CASE C: When MCC[4:0]=0x1F
    - ASBWA[10:7] = Antenna[3:0] increments by one
    - ASBWA[2] = HalfChip must be cleared
  - CASE D: When MCC[4:0]=0x0F
    - ASBWA[10:7] = Antenna[3:0] increments by one
    - ASBWA[2] = HalfChip must be set
  - CASE E: Otherwise
    - ASBWA[10:7] = Antenna[3:0] must be incremented by one

Antenna Sample Buffer Write Address Generation Implementation

- To implement the above five cases, let us define two registers:
  - REGA = 128 - 4 = 124 = 0x7C
  - REGB = 128 = 0x80

- CASE A: This state occurs when the entire Antenna Sample Buffer has been written and the address field must be cleared so that the buffer may start again at the beginning address
- MCC[11:0]=0xFFF
  - ASBWA[10:7] = Antenna[3:0] must be cleared
  - ASBWA[6:3] = ChipCount[6:3] must be cleared
  - ASBWA[2] = HalfChip must be cleared
    - ALUB = shifterconst=0x0
    - ALU = PASSB

Antenna Sample Buffer Write Address Generation Implementation

- CASE B: This state occurs when both a chip and HalfChip have been written to each of the eight banks and the address field must point back to the first bank
- MCC[7:0]=0xFF
  - ASBWA[10:7] = Antenna[3:0] increments by one
  - ASBWA[6:3] = ChipCount[6:3] increments by one
  - ASBWA[2] = HalfChip must be cleared
    - Since we know ChipCount[6:3] ≠ 0xF, incrementing ChipCount[6:3] will not generate a carry into Antenna[3:0]
    - Since we know HalfChip=1, we can increment HalfChip and it will toggle HalfChip AND generate a carry to increment ChipCount[6:3]
    - ALUA = ALUOUTREG
    - ALUB = 128 + 4 = REGB OR shifterconst=0x4
    - ALU = ALUA + ALUB

Antenna Sample Buffer Write Address Generation Implementation

- CASE C: This state occurs when after a sample (HalfChip=1) has been written to a single bank for all twelve antennas and the address field must point to the next memory bank
- MCC[4:0]=0x1F
  - ASBWA[10:7] = Antenna[3:0] increments by one
  - ASBWA[2] = HalfChip must be cleared
    - ALUA = 128 - 4 = REGA
    - ALUB = ALUOUTREG
    - ALU = ALUA + ALUB

Antenna Sample Buffer Write Address Generation Implementation

- CASE D: This state occurs when after a sample (HalfChip=0) has been written to a single bank for all twelve antennas and the address field must point to the next half-chip address within the same memory bank
- MCC[4:0]=0x0F
  - ASBWA[10:7] = Antenna[3:0] increments by one
  - ASBWA[2] = HalfChip must be set
    - ALUA = ALUOUTREG
    - ALUB = 128 + 4 = REGB OR shifterconst=0x4
    - ALU = ALUA + ALUB

Antenna Sample Buffer Write Address Generation Implementation

- CASE E: This state occurs when the conditions for any of the cases A through D are not met and the address field must point to the next antenna sample
  - ASBWA[10:7] = Antenna[3:0] must be incremented by one
    - ALUA = ALUOUTREG
    - ALUB = 128 = REGB
    - ALU = ALUA + ALUB
- Note that for both CASE B and CASE D, the instructions to the DPU are identical
- We therefore only have four unique states for the DPU

Antenna Sample Buffer Write Address Generation Implementation

- Let us define the following four states
  - State 00: DPU instruction for CASE A
  - State 01: DPU instruction for CASE B and CASE D
  - State 10: DPU instruction for CASE C
  - State 11: DPU instruction for CASE E
- Let us use the notation of A when case A is true and /A when CASE A is not active, and X represents don't care
- We also must assign priority to the five cases since the priority is base on the MCC[11:0] higher order bits having higher priority
  - Priority 1: A , CASE B=X, C=X, D=X E=X — State 00
  - Priority 2: (!A & B), C=X, D=X E=X — State 01
  - Priority 3: (!A & !B & C & !D) — State 10
  - Priority 3: (!A & !B & !C & D) — State 01
    - Note CASE C and CASE D have equal priority
  - Priority 4: !A & !B & !C & !D — State 11
    - Note that the encoding !A & !B & C & D is not physically possible

Antenna Sample Buffer Write Address Generation Implementation

- Let Pn represent the priority for priority with level n
- We can fill in the priority assignments for the Karnaugh map entries for the sixteen possibilities for P0, P1, P2, and P3

PRIORITY ASSIGNMENTS

		D=1 C=1			
		00	01	11	10
B=1	00	P4	P3	X	P3
	01	P2	P2	P2	P2
A=1	11	P1	P1	P1	P1
	10	P1	P1	P1	P1

Antenna Sample Buffer Write Address Generation Implementation

If we fill in the DPU state tables with the state assignments of the previous page:

Priority 1: A , CASE B=X, C=X, D=X E=X — State 00
Priority 2: (!A & B), C=X, D=X E=X — State 01
Priority 3: (!A & !B & C & !D) — State 10
Priority 3: (!A & !B & !C & D) — State 01
Note CASE C and CASE D have equal priority
Priority 4: !A & !B & !C & !D — State 11
Note that the encoding !A & !B & C & D is not physically possible

STATE[1]

B=1 / A=1 \ D=1 C=1	00	01	11	10
00	1	0	X	1
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

STATE[0]

B=1 / A=1 \ D=1 C=1	00	01	11	10
00	1	1	X	0
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

STATE[1] = !A & !B & !D
STATE[0] = (!A & B) | (!B & !C)

Note that we do not need to compute variable D

Antenna Sample Buffer Write Address Generation Implementation

In order to be able to decode the conditions A, B, C, and D and their inverses, without using excessive product terms, we decode a count one before the desired count and register the result

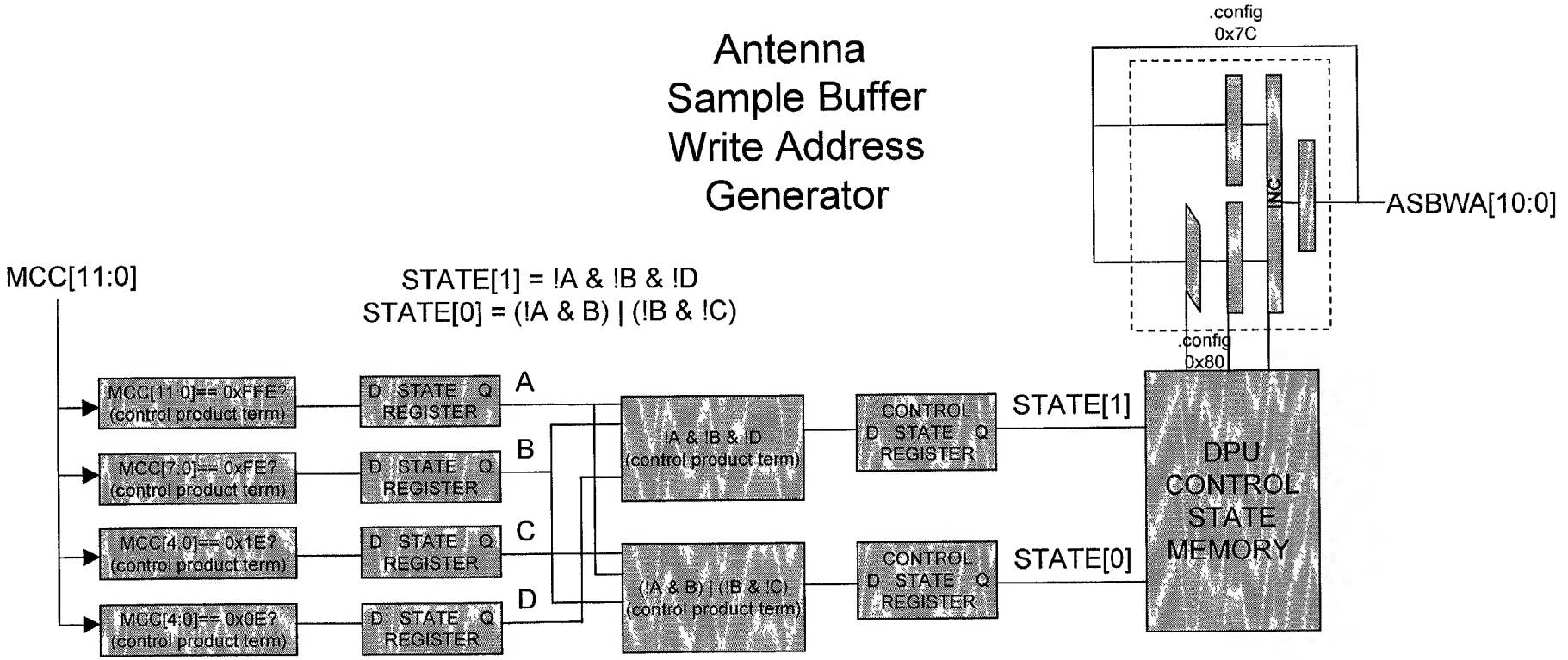
REGA: MCC[11:0] = 0xFFE = MCC[11] & MCC[10] & MCC[9] & MCC[8] & MCC[7] & MCC[6] & MCC[5] & MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

REGB: MCC[7:0] = 0xFE = MCC[7] & MCC[6] & MCC[5] & MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

REGC: MCC[4:0] = 0x1E = MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

REGD: MCC[4:0] = 0x0E = !MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

Antenna Sample Buffer Write Address Generator Implementation



Antenna Sample Buffer Bank Write Enable Generation

- Each of the eight banks in the Antenna Sample Buffer contains the samples eight chips apart for all twelve antennas.
- The control must generate a Antenna Sample Buffer Write Enable ($ASBWE_n$) signal for each of the n banks
- The timing for the $ASBWE_n$ signals is based upon the Master Chip Counter bits MCC[7:0]
- Note that the MCC counter implicitly counts through values for sixteen antennas, but the write enable signals are only enabled during the first twelve

Antenna Sample Buffer Bank Write Enable Generation

- $ASBWE_n$ is active when MCC[7:5] = ChipCount[2:0] matches the addressed memory bank and MCC[3:0] = Antenna[3:0] addresses antennas 0 to 11:

  - $ASBWE_0$ = !MCC[7] & !MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
    $ASBWE_1$ = !MCC[7] & !MCC[6] & MCC[5] & !(MCC[3] & MCC[2])
  - $ASBWE_2$ = !MCC[7] & MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
  - $ASBWE_3$ = !MCC[7] & MCC[6] & MCC[5] & !(MCC[3] & MCC[2])
  - $ASBWE_4$ = MCC[7] & !MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
  - $ASBWE_5$ = MCC[7] & !MCC[6] & MCC[5] & !(MCC[3] & MCC[2])
  - $ASBWE_6$ = MCC[7] & MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
  - $ASBWE_7$ = MCC[7] & MCC[6] & MCC[5] & !(MCC[3] & MCC[2])

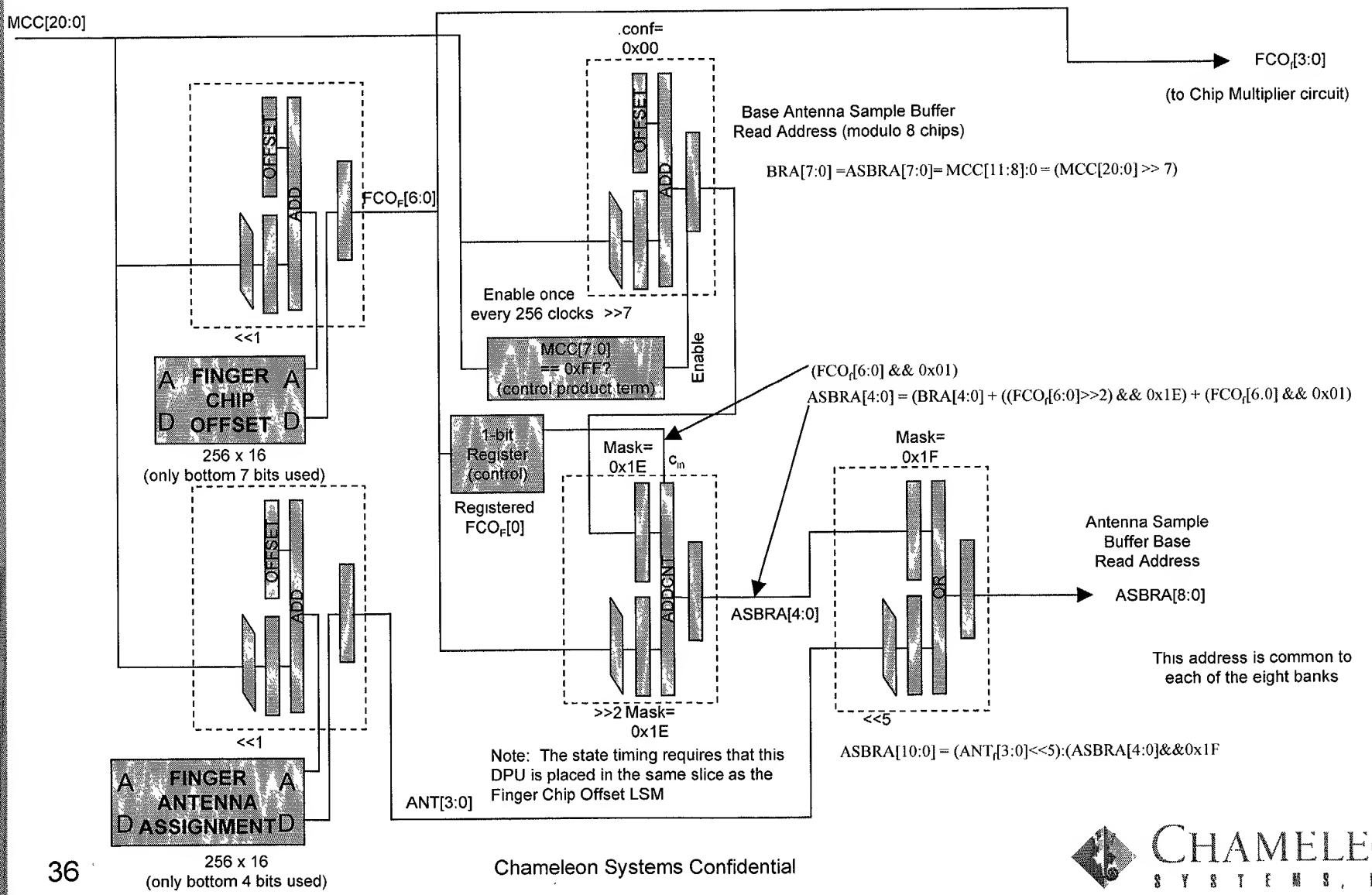
Antenna Sample Buffer Read Address Generation (1 of 2)

- Compute the Antenna Sample Buffer Base Read Address (ASBRA) for the Antenna Sample Buffer as follows:
  - The Base Read Address (BRA) is the fixed offset between the Antenna Sample Buffer Write Address and the Antenna Sample Buffer Read Address (ASRA) for zero finger chip offset
  - For each of the 256 fingers:
    - Add the Finger Chip Offset (FCO_f) to the BRA
    - Merge the Finger Antenna Assignment (ANT_f) bits to the ASBRA
    - The Antenna Sample Buffer Read Address (ASBA) is formed by shifting the ASBRA left two bit positions

Antenna Sample Buffer Read Address Generation (2 of 2)

- Compute the Base Read Address (BRA):
  - BRA[4:0] =ASBRA[4:0]= MCC[11:8]:0 = ((MCC[20:0] >> 7) && 0x1E)
    - The Base Read Address is latched to remain constant for 256 clocks
    - An offset can be added if necessary to adjust timing
- For each of the f fingers (0-255)
  - Read the antenna assignment (ANT_f) for the current finger
  - Read the Finger Chip Offset (FCO_f) for the current finger
  - Add the Finger Chip Offset to the Base Address Register
    - ASBRA[4:0] = (BRA[4:0] + ((FCO_f[6:0]>>2) && 0x1E) + (FCO_f[6:0] && 0x01)
  - The Antenna Assignment ANT_f[3:0] is placed in bits ASBRA[8:5]
    - ASBRA[8:0] = (ANT_f[3:0]<<5):ASBRA[4:0]
  - The Antenna Sample Buffer Read Address (ASRA) is formed by shifting ASBRA left two bit positions
    - ASRA[10:0] = ASBRA[8:0] << 2

Antenna Sample Buffer Base Read Address Generator Implementation



Antenna Sample Buffer Read Address Offset Circuit

- The Antenna Sample Buffer Base Base Read Address, (BRA) is on a multiple of eight sample boundary
- If the Finger Chip Offset for a finger, FCO_f, is not on a multiple-of-eight sample boundary, then the correct eight consecutive samples are not output on the memory
- To output the correct words, some of the addresses must not select the word specified by the ASBRA, but the next full sample into the buffer
- $FCO_f[3:1]$ are used to determine which of the eight Antenna Sample Buffer address generators need to have their Bankn_ReadWordPlus8 asserted in order to read the next eighth sample into the buffer

Antenna Sample Buffer Read Address Offset Circuit

- The ASBRA calculation requires that the Finger Chip Offset for a given finger, FCO_f, is read out of the FCO memory before it is needed for the Read Address Offset Circuit
  - $FCO_f[0]$ is needed one cycle after it is read from the FCO memory, so the DPU using $FCO_f[0]$ as a control input must be placed in the same tile as the FCO memory, because $FCO_f[0]$ is delayed one cycle through the Control State Memory
  - $FCO_f[3:1]$ is needed by the Read Address Offset Circuit two cycles after it is read from the FCO memory
  - An additional delay of one cycle for $FCO_f[3:1]$ is achieved if $FCO_f[3:1]$ is routed through Broadcast Bit horizontal long lines and the following equations are used:

Antenna Sample Buffer Bankn_ReadWordPlus8 Generation

Case FCOf[3:1]

0: Bank0_ReadWordPlus8 = 0
Bank1_ReadWordPlus8 = 0
Bank2_ReadWordPlus8 = 0
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0

1: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 0
Bank2_ReadWordPlus8 = 0
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0

2: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 0
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0

3: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0

4: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0

5: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 1
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0

6: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 1
Bank5_ReadWordPlus8 = 1
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0

7: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 1
Bank5_ReadWordPlus8 = 1
Bank6_ReadWordPlus8 = 1
Bank7_ReadWordPlus8 = 0

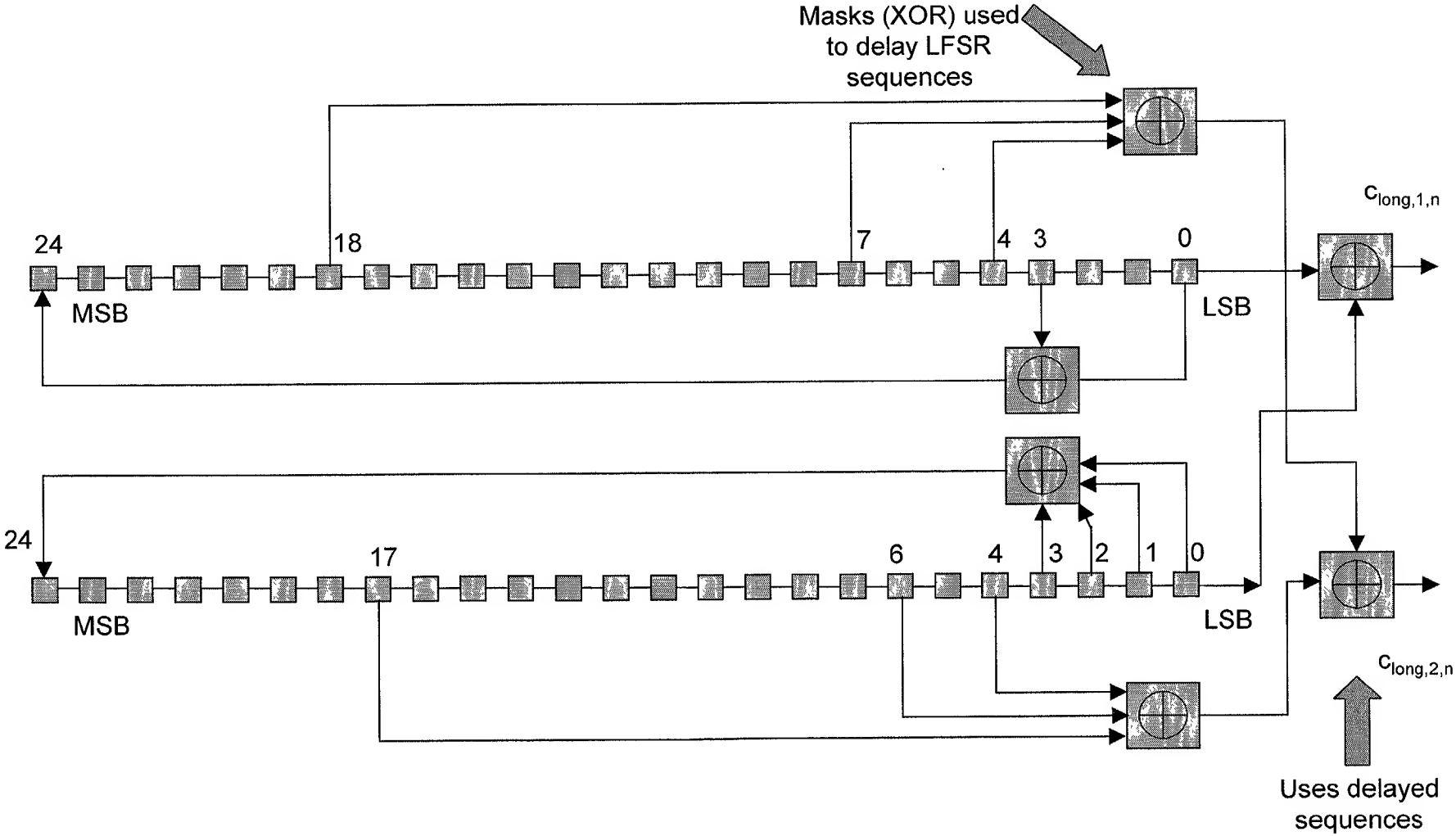
Antenna Sample Buffer Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - 22 DPUs
  - 26 LSMs

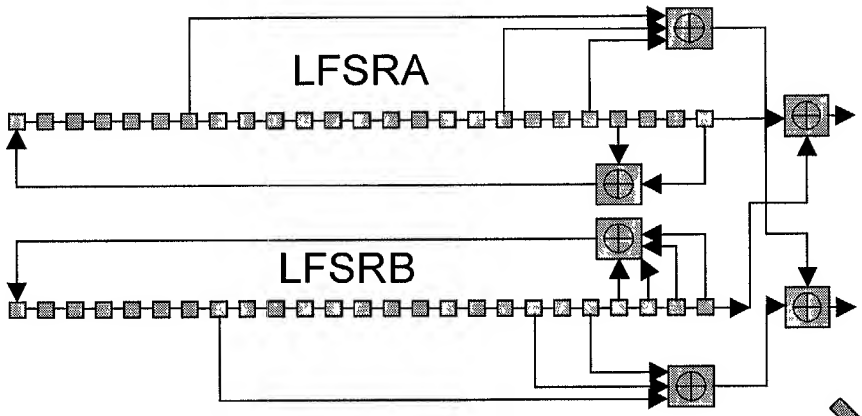
UMTS Gold Code Generator Requirements & Assumptions

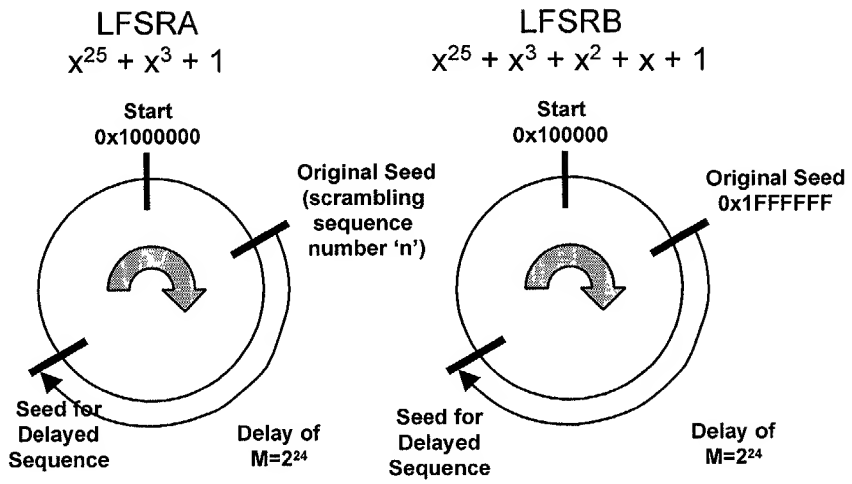
- Requirements
  - Needs to generate 16 bits per clock (8I, 8Q)
  - 32 Users at 125 MHz (The generator supports 64 users @125 MHz)
  - Easily scale to 64 users @ 250 MHz with single engine
  - 32-64 user expansion is only a function of memory
- Assumptions
  - The same Gold Code output may be shared by the different fingers of a single channel if the fingers are aligned
  - Each Gold Code is unique per user

UMTS Gold Code Generator as defined by 3G TS 25.213



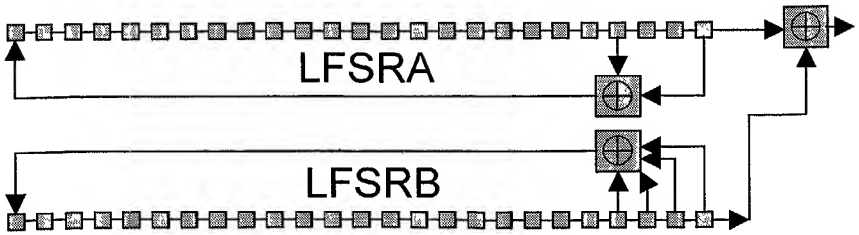
Exploit Code Properties to Eliminate Masks



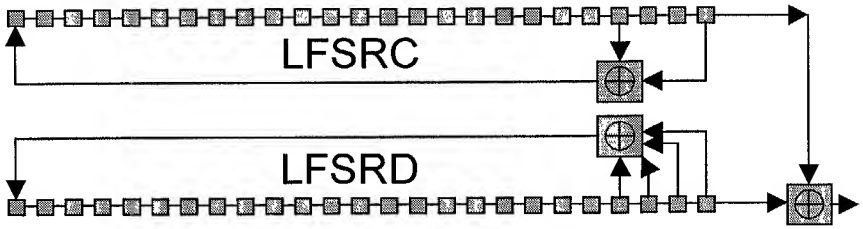


Implementation Unchanged

Use second pair of seeds to eliminate masks (avoid XOR implementation)



LFSR Pair to Calculate $c_{long,1,n}$

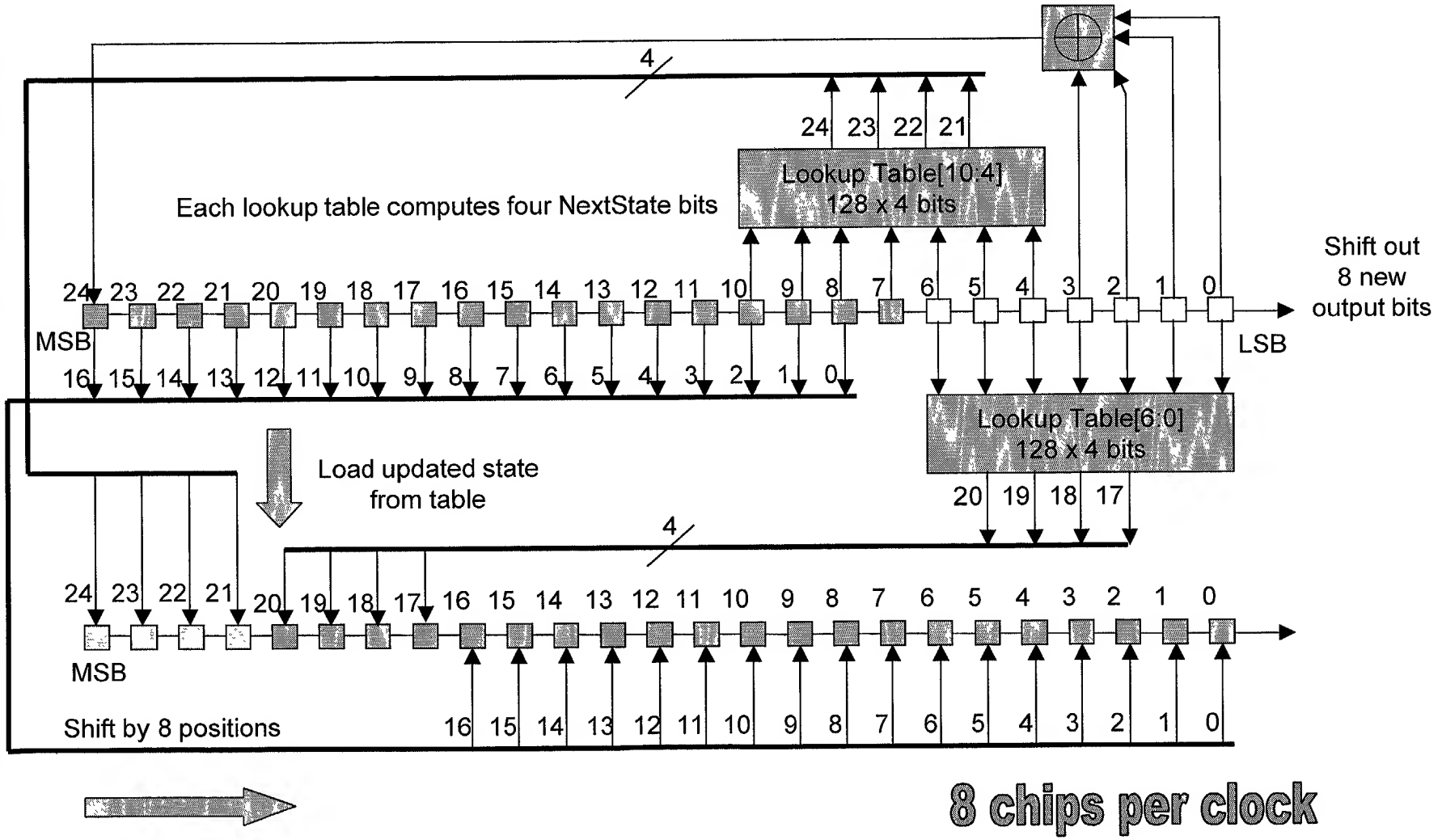


LFSR Pair to Calculate $c_{long,2,n}$

Linear Feedback Shift Register (LFSR) Initial Values

- Each LFSR is reset to its initial value at the beginning of the frame
- The seed for LFSRA is assigned by the Network Controller
- The seed for LFSRB is 0x1FFFFFF for all users
- The seed for LFSRC is the contents of LFSRA's seed shifted by 16,777,232 cycles, and is computed by the ARC at the beginning of the call
- The seed for LFSRD is 0x1FFFFFF shifted by 16,777,232 cycles for all users and is static
- The seed values for all LFSRs are stored in LSMs

Lookup Table Determines the Next 8 Bits In a Single Cycle



8 chips per clock

CHAMELEON SYSTEMS, INC.

Gold Code Generator Output Computations

$C_{long1,n}$ = LSFRA[7:0] XOR LSFRB[7:0]
Let us define LFSRC'[i] = LSFRC[2⌊i/2⌋]

$C_{long,n}(i) = C_{long1,n}(i)(1 + j(-1)^{i}(c_{long2},n(2\lfloor i/2 \rfloor))$ (from 3G TS25.213)

Multiplying bits by +1/-1 is the same as XOR for 0s and 1s.
XORing by 0xAA can be used in place of the $(-1)^{i}$ term.
In binary representation, the Scrambling Code $C_{long,n}$ becomes:

$C_{long,n}[7:0] = C_{long1,n}[7:0](1 + j(0xAA) \text{ XOR } C_{long2,n}[7:0])$

$C_{long,n}[7:0]$ = LFSRA[7:0] XOR LFSRB[7:0]
+j(LFSRA[7:0] XOR LFSRB[7:0] XOR 0xAA XOR LFSRC'[7:0] XOR LFSRD'[7:0]

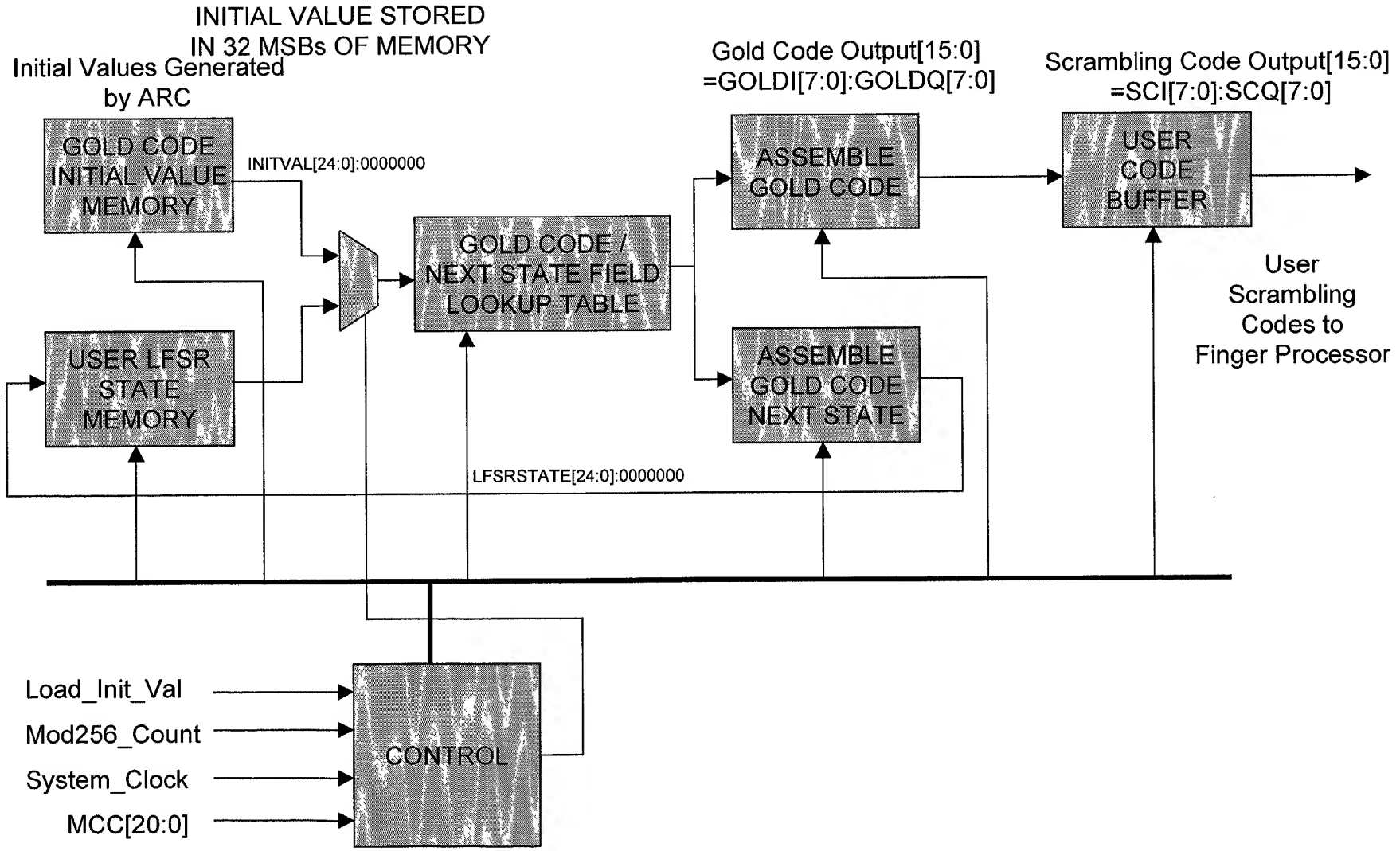
$C_{long,n}[7:0]$ = SCI[7:0] + jSCQ[7:0]

Let us define LFSRD''[7:0] = 0xAA XOR LFSRD'[7:0], then:

$C_{long,n}[7:0]$ = (LFSRA[7:0] XOR LFSRB[7:0])
+ j(LFSRA[7:0] XOR LFSRB[7:0] XOR LFSRC'[7:0] XOR LFSRD''[7:0])

We use a lookup table to compute LFSRC'[7:0] and LFSRD''[7:0])

Gold Code Generator Functional Block Diagram



Gold Code Generator Memory Layout

Gold Code Initial Value Memory

Address	Contents
0xFC	User 63 LFSRD
0xF8	User 63 LFSRC
0xF4	User 63 LFSRB
0xF0	User 63 LFSRA
0xEC	User 62 LFSRD
0xE8	User 62 LFSRC
0xE4	User 62 LFSRB
0xE0	User 62 LFSRA
	⋮
0x1C	User 1 LFSRD
0x18	User 1 LFSRC
0x14	User 1 LFSRB
0x10	User 1 LFSRA
0x0C	User 0 LFSRD
0x08	User 0 LFSRC
0x04	User 0 LFSRB
0x00	User 0 LFSRA

User LFSR State Memory

Address	Contents
0xFC	User 63 LFSRD
0xF8	User 63 LFSRC
0xF4	User 63 LFSRB
0xF0	User 63 LFSRA
0xEC	User 62 LFSRD
0xE8	User 62 LFSRC
0xE4	User 62 LFSRB
0xE0	User 62 LFSRA
	⋮
0x1C	User 1 LFSRD
0x18	User 1 LFSRC
0x14	User 1 LFSRB
0x10	User 1 LFSRA
0x0C	User 0 LFSRD
0x08	User 0 LFSRC
0x04	User 0 LFSRB
0x00	User 0 LFSRA

User Code Buffer Memory

Address	Contents
0x1FE	User 63 PONG
0x1FC	User 62 PONG
0x1FA	User 61 PONG
	⋮
0x104	User 2 PONG
0x102	User 1 PONG
0x100	User 0 PONG
0xFE	User 63 PING
0x0C	User 62 PING
	⋮
0x04	User 2 PING
0x02	User 1 PING
0x00	User 0 PING

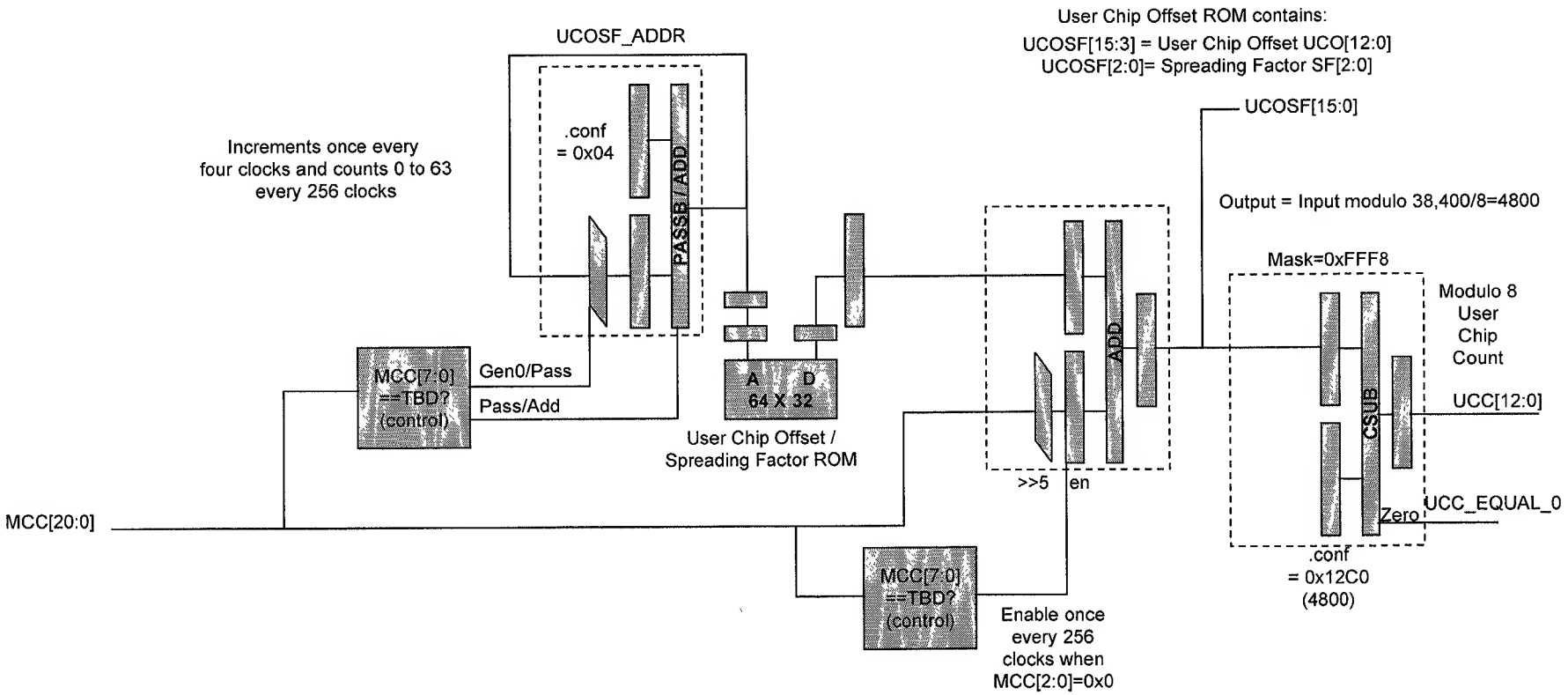
Gold Code Generator Lookup[6:0] Definitions

At Address 4n+0: OUT[7:0] = Next StateA[3:0]:PASSA[3:0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]	At Address 4n+2: OUT[7:0] = Next StateC[3:0]:LFSRC'[3:0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0]
At Address 4n+1: OUT[7:0] = Next StateB[3:0]:PASSB[3:0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]	At Address 4n+3: OUT[7:0] = Next StateD[3:0]:LFSRD''[3:0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = /IN[2] OUT[2] = IN[2] OUT[1] = /IN[0] OUT[0] = IN[0]

Gold Code Generator Lookup[10:4] Definitions

At Address 4n+0: OUT[7:0] = IN[7:4]:Next StateA[7:4] OUT[7] = IN[3] OUT[6] = IN[2] OUT[5] = IN[1] OUT[4] = IN[0] OUT[3] = IN[6] XOR IN[3] OUT[2] = IN[5] XOR IN[2] OUT[1] = IN[4] XOR IN[1] OUT[0] = IN[3] XOR IN[0]	At Address 4n+2: OUT[7:0] = IN'[7:4]:Next StateC[7:4] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0]
At Address 4n+1: OUT[7:0] = IN[7:4]:Next StateB[7:4] OUT[7] = IN[3] OUT[6] = IN[2] OUT[5] = IN[1] OUT[4] = IN[0] OUT[3] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[2] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[1] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[0] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0]	At Address 4n+3: OUT[7:0] = IN''[7:4]:Next StateD[7:4] OUT[3] = /IN[2] OUT[2] = IN[2] OUT[1] = /IN[0] OUT[0] = IN[0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0]

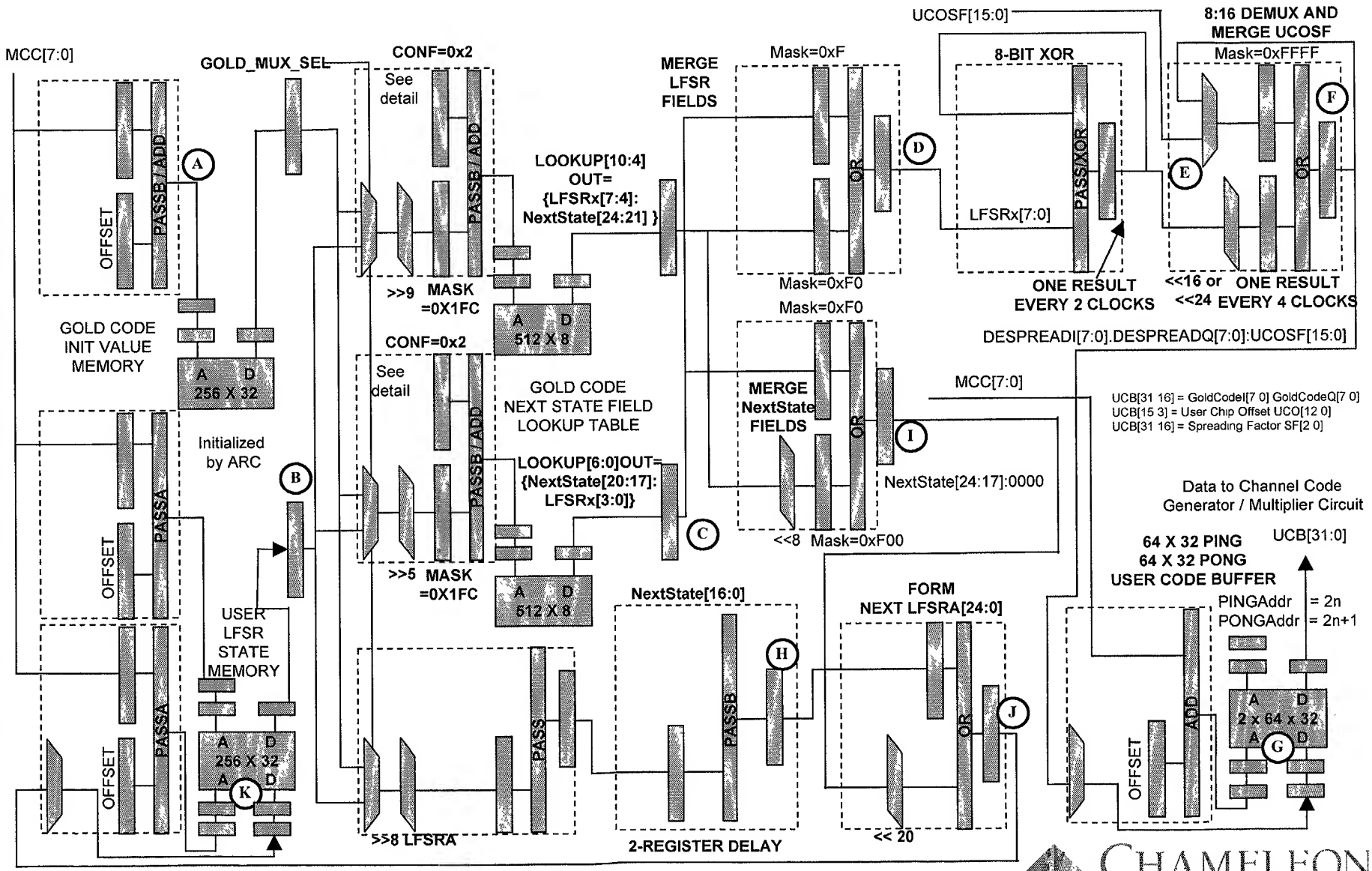
Gold Code Control Implementation



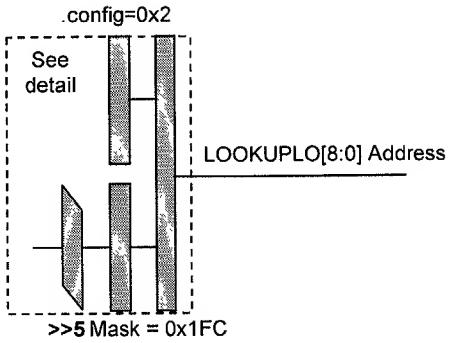
Add User Chip Offset
to Master Chip Counter

```
If MCC[7:0]= TBD    ;Address=0
  ALUB = shifterconstant = 0
  ALU = PASSB
else      ;Address = Address + 4
  ALUB = BREG
  ALU = ADD
```

Gold Code Generator Implementation



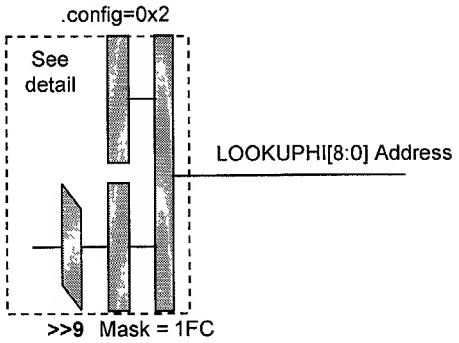
Gold Code Generator Implementation
Lookup[6:0] Address Generation DPU



Control needs to be able to select one of four lookup
tables in the RAM.
Each lookup table is selected by two control inputs Select[1:0]:
Select[1:0] resides in the bottom two bits of the address
If Select[1:0]=0x0 then
  ALU=PASSB
  Address[1:0]=0x0
  Output[7:4]= NextStateA[20:17], Output[3:0]=LFSRA[3:0]
If Select[1:0]=0x1 then
  ALU=INC (B+1)
  Address[1:0]=0x1
 Output[7:4]= NextStateB[20:17], Output[3:0]=LFSRB[3:0]
If Select[1:0]=0x2 then
  ALU=ADD (A + B)
  Address[1:0]=0x2
 Output[7:4]= NextStateC[20:17], Output[3:0]=LFSRC'[3.0]
If Select[1:0]=0x3 then
  ALU=ADDCNT (A + B + 1, force Cin by control)
  Address[1:0]=0x3
 Output[7:4]= NextStateD[20:17], Output[3:0]=LFSRD''[3:0]

Select[1:0] is a function of the Master Chip Counter bits MCC[1:0]

Gold Code Generator Implementation Lookup[10:4] Address Generation DPU



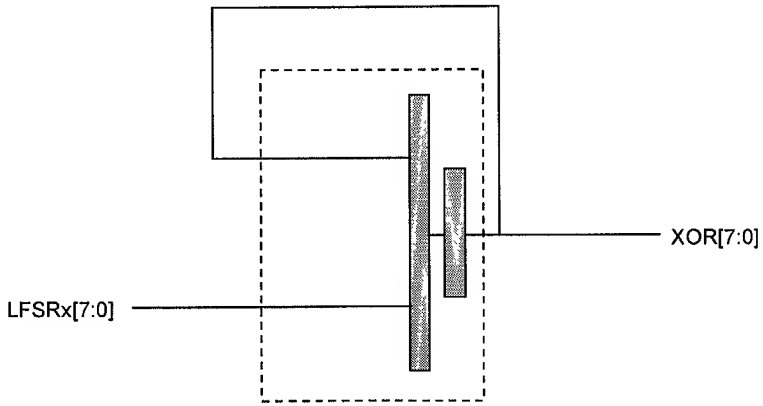
Control needs to be able to select one of four lookup
tables in the RAM.
Each lookup table is selected by two control inputs Select[1:0]:
Select[1:0] resides in the bottom two bits of the address
If Select[1:0]=0x0 then
  ALU=PASSB
  Address[1:0]=0x0
  Output[7:4]= LFSRA[7:4], Output[3:0]= NextStateA[24:21]
If Select[1:0]=0x1 then
  ALU=INC (B+1)
  Address[1:0]=0x1
  Output[7:4]= LFSRB[7:4], Output[3:0]= NextStateB[24:21]
If Select[1.0]=0x2 then
  ALU=ADD (A + B)
  Address[1:0]=0x2
  Output[7:4]= LFSRC'[7:4], Output[3:0]= NextStateC[24:21]
If Select[1:0]=0x3 then
  ALU=ADDCNT (A + B + 1, force C_{in} by control)
  Address[1:0]=0x3
  Output[7:4]= LFSRD"[7:4], Output[3:0]= NextStateD[24:21]

Select[1:0] is a function of the Master Chip Counter bits MCC[1:0]

Gold Code Generator Implementation
8-Bit XOR DPU

D	LFSR Input	A	B	C	D	A	B	C	D	A	B
	XOR DPU ALU Instruction	PASSB	XOR	XOR	XOR	PASSB	XOR	XOR	XOR	PASSB	XOR
E	XOR Output Register	A^B^C^D	A	A^B	A^B^C	A^B^C^D	A	A^B	A^B^C	A^B^C^D	A

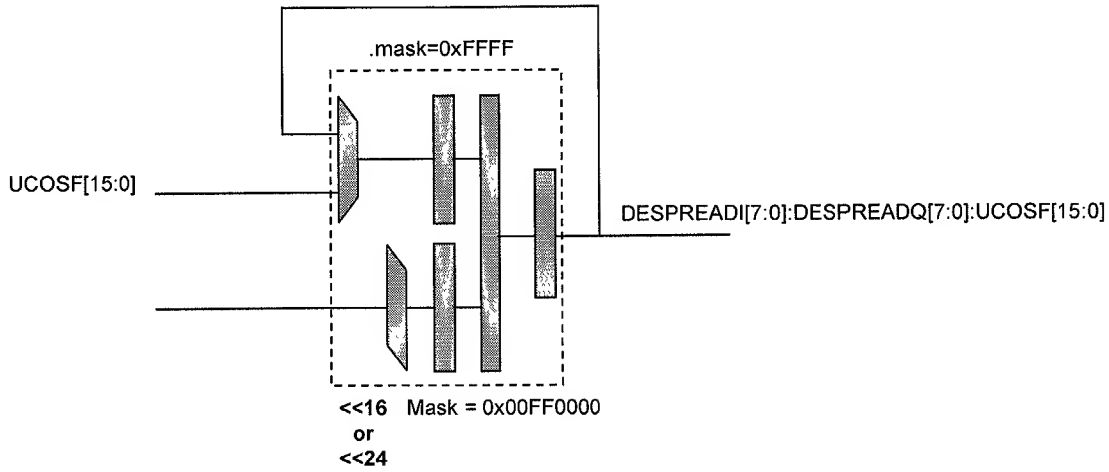
Where:
A = LFSRA[7:0]
B = LFSRB[7:0]
C = LFSRC'[7:0]
D = LFSRD'[7:0]
^ = XOR



This DPU is used to perform an XOR operation

Control needs to generate two states for the DPU:
If PASS
;Pass the input to the output register
ALUB=LFSRx[7:0] (from previous stage)
ALU = PASSB
OUTREGEN = 1
Else
;XOR the input with the output register
ALUA = OUTREG
ALUB=LFSRx[7:0] (from previous stage)
ALU = XOR
OUTREGEN = 1

Gold Code Generator Implementation 8:16 Demultiplex and Merge UCOSF DPU

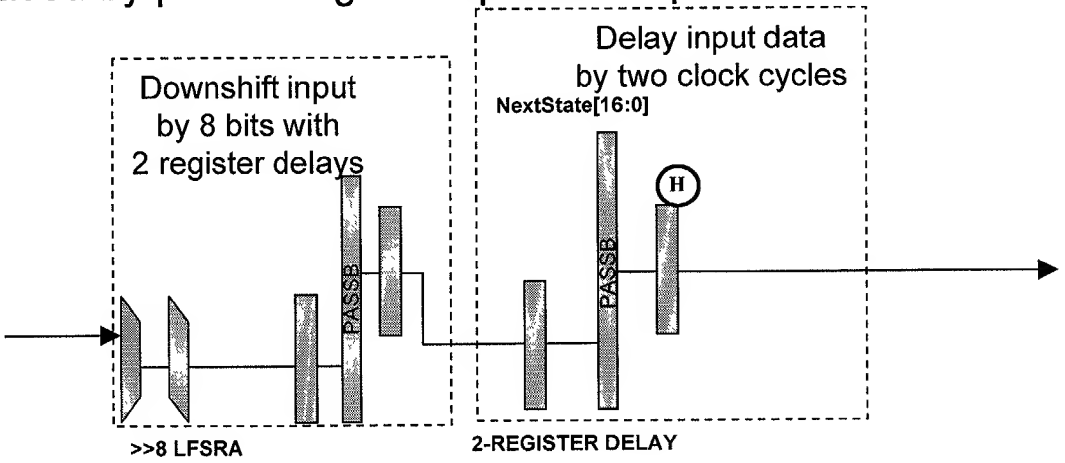


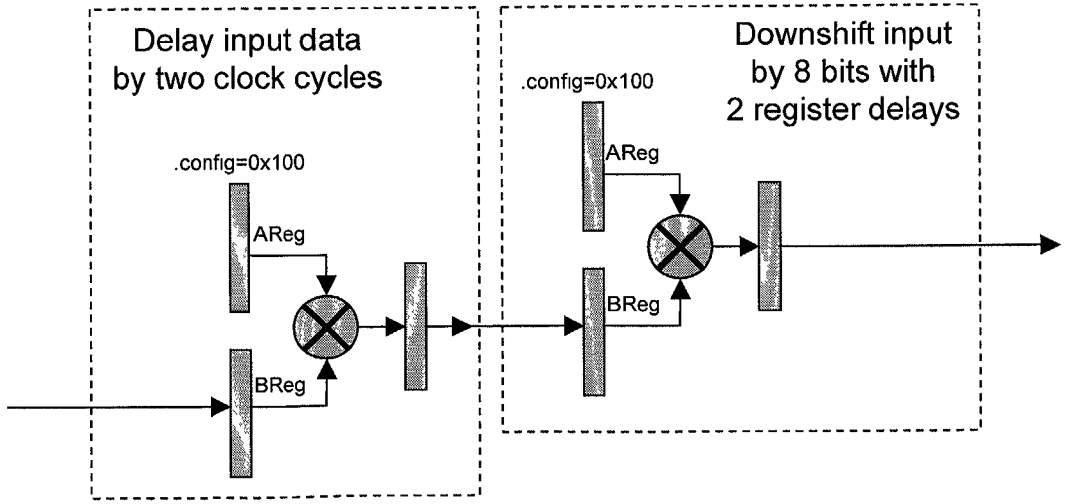
This DPU is used to perform a 8:16 demultiplexing operation on the LSFR data as well as merge the UCOSF[15:0] field into the data stream before it is written into the User Code Buffer RAM.

Control needs to generate three states for the DPU:
If MERGE_I
  ;Place UCOSF[15:0] in bits [15:0]
  ;and place DESPREADI[7:0] in bits [31:24]
  ALUA=UCOSF[15:0] && (Mask==0xFFFF)
  ALUB=(DESPREADI[7:0] from previous stage)<<24
  ALU = OR
  OUTREGEN = 1
If MERGE_Q
  ;Merge DESPREADQ[7:0] into bit positions [23:16]
  ALUA = OUTREG
  ALUB =((DESPREADQ[7.0] from previous stage)<<16) && Mask==0x00FF0000
  ALU = OR
  OUTREGEN = 1
Else
  OUTREGEN = 0

Gold Code Generator Implementation of Downshift Pipeline Delay with Multipliers

Two DPUs may be saved by performing the equivalent operations with two multipliers





Gold Code Generator Timing

A	Mod256_Count	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D	U3A	U3B	U3C	U3D	U4A	U4B	U4C	U4D
B	User LFSR State Output	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D	U3A	U3B	U3C	U3D
C	Gold Code Lookup Output	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D
D	Merge LFSRx[7:0] Fields	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C
E	XOR LFSRx[7:0] Fields		U61Q		U62I		U62Q		U63I		U63Q		U0I		U0Q		U1I		U1Q		U2I
F	Merge Scrambling Codes			U61IQ				U62IQ				U63IQ				U0IQ				U1IQ	
G	User to Finger Buffer Input	U60IQ				U61IQ				U62IQ				U63IQ				U0IQ			
H	Form NextState LFSRx[16:0]	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D
I	Merge NextState Fields	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C
J	Form NextState LFSRx[24:0]	U61C	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B
K	User LFSR State Memory Input	U61A	U61B	U61C	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D

Gold Code Data Format:
Gold Code Output[15:0] =GOLDI[7:0]:GOLDQ[7:0]

Gold Code Generator
UCOSF Output Timing

	UCOSF[15:0]	U0	U0	U62	U62	U1	U1	U63	U63	U2	U2	U0	U0	U3	U3	U1	U1	U4	U4	U2	U2
	UCC[12:0], UCC_EQUAL_0	XX	U0	U0	XX	XX	U1	U1	XX	XX	U2	U2	XX	XX	U3	U3	XX	XX	U4	U4	XX
	REG_UCC_EQUAL_0	XX	XX	U0	U0	U0	U0	U1	U1	U1	U1	U2	U2	U2	U2	U3	U3	U3	U3	U4	U4
B	GOLD_MUX_SEL	U63	U63	U63	U0	U0	U0	U0	U1	U1	U1	U1	U2	U2	U2	U2	U3	U3	U3	U3	U4
E	Gold Code 8-BIT XOR	U61B		U62A		U62B		U63A		U63B		U0A		U0B		U1A		U1B		U2A	
E	UCOSF[15:0]			U62				U63				U0				U1				U2	
F	8:16 Demux Output		U61		U62 temp		U62		U63 temp		U63		U0 temp		U0		U1 temp		U1		U2 temp

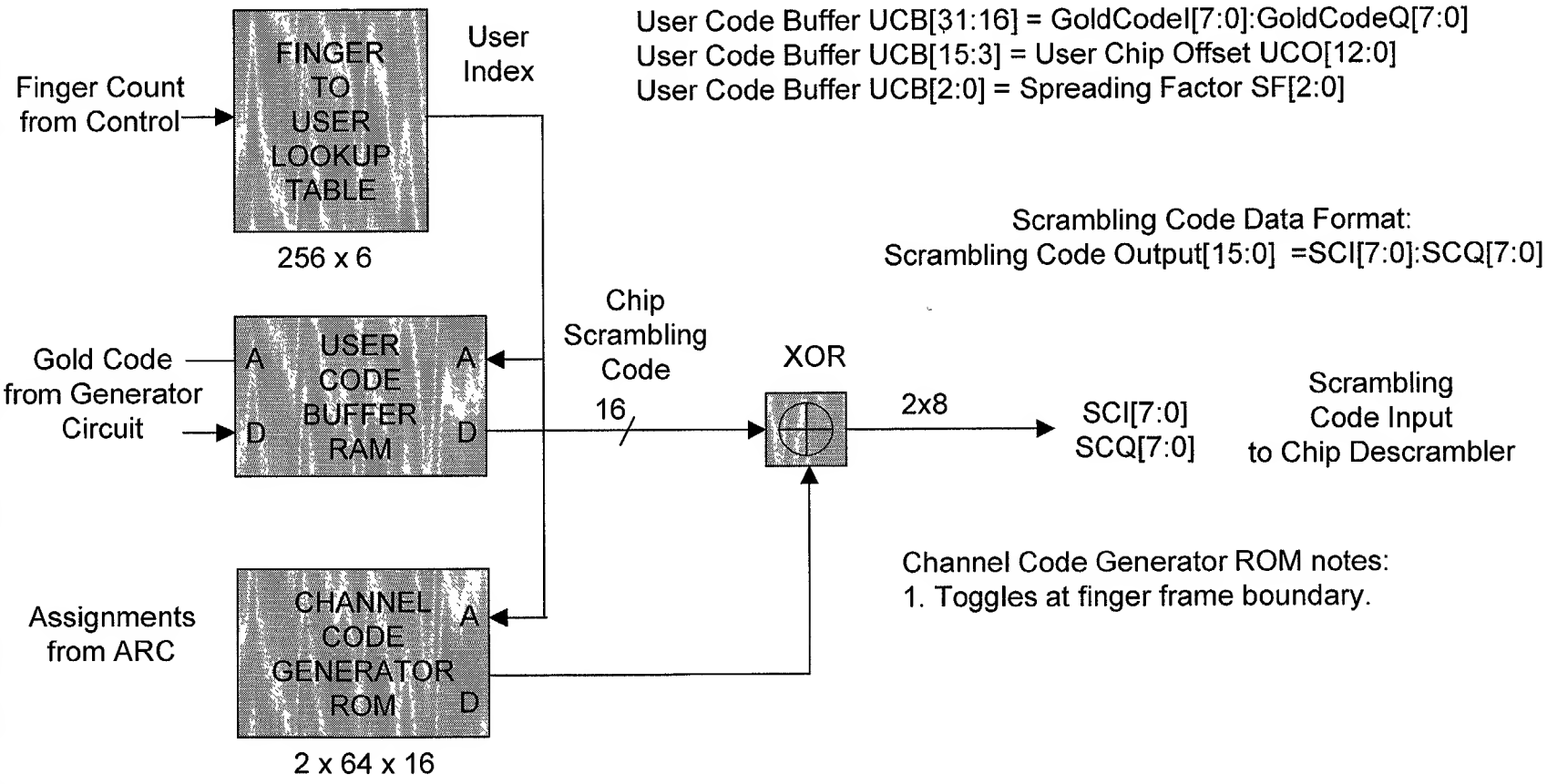
Gold Code Generator Resource Requirements

- 32 User Implementation @ 125 MHz
  - 14 DPUs
  - 8 LSMs
  - 2 Multipliers
- 64 User Implementation @ 125 MHz
  - 14 DPUs
  - 10 LSMs
  - 2 Multipliers
- 128 User Implementation @ 250 MHz
  - 16 DPUs
  - 12 LSMs
  - 2 Multipliers

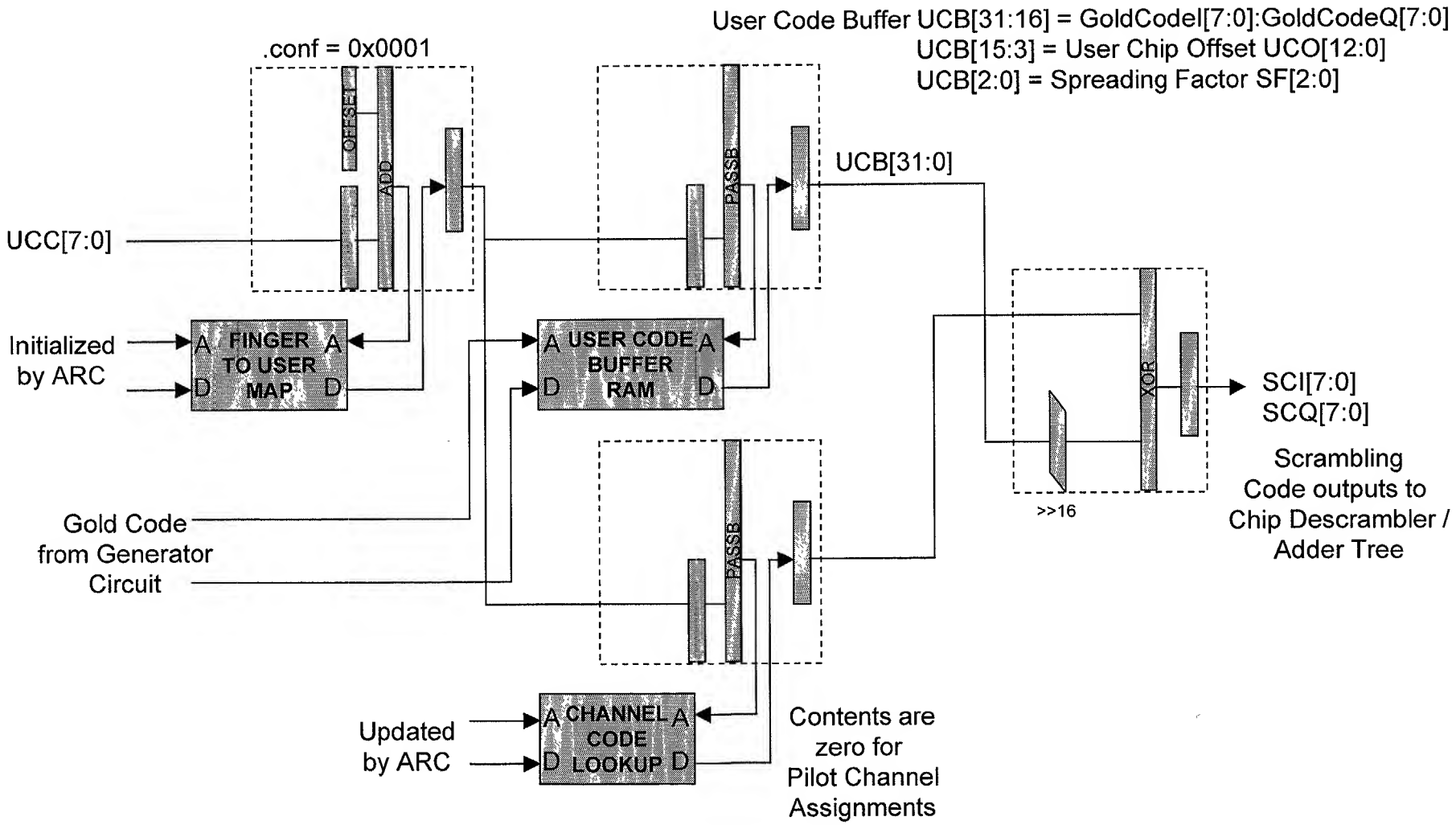
Channel Code Generator / Multiplier Requirements and Assumptions

- Requirements
  - Provide a User to Finger Interface
  - Convert between User-based Gold Code and Finger-based Chip despreading
- Assumptions (from 3G TS25.213 Specification)
  - DPCCH $c_c = c_{ch,256,0} = 1, 1, 1, 1, \ldots$ (all ones)
  - Single DPDCH $c_{d,1} = c_{ch,SF,k}$ where k=SF/4
    - $c_{ch,4,1} = 1, 1, -1, -1, \ldots$
    - $c_{ch,8,2} = 1, 1, -1, -1, \ldots$
    - $c_{ch,16,4} = 1, 1, -1, -1, \ldots$
  - If more than one $DPDCH_n$ SF=4, $c_{d,n} = c_{ch,4,k}$
    - k=1 if $n \in \{1,2\}$
    - k=3 if $n \in \{3,4\}$
    - k=2 if $n \in \{5,6\}$

Channel Code Generator / Multiplier Functional Block Diagram



Channel Code Generator / Multiplier Implementation



Channel Code Generator / Multiplier Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - ◆ 4 DPUs
  - ◆ 2 LSMs
- Note that the Gold Code Output Buffer LSM resources are counted in the Gold Code Generator circuit

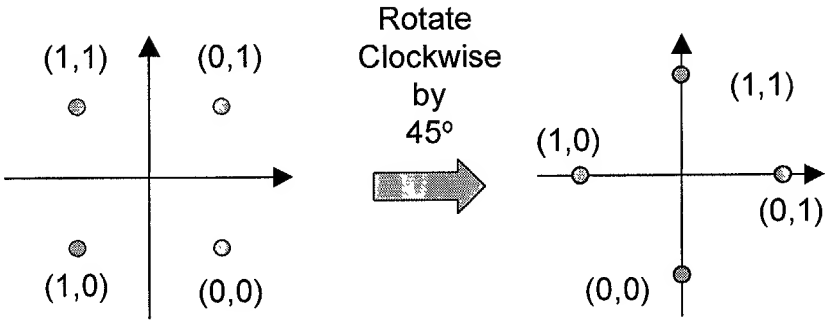
Chip Descrambler Requirements and Assumptions

- Requirements
  - Include the Adder Tree Input multiplexer
  - Descramble 8 chips per clock
  - Process 256 fingers @ 125 MHz
  - Process 512 fingers @ 250 MHz
- Assumptions
  - I must be able to read 8 consecutive samples (T_c apart) per clock from the Antenna Sample Buffer from any one antenna
  - An 8:1 Multiplexer is needed to align the input data to an even SF boundary point
  - All fingers with SF=4 will require 2 consecutive finger assignments so that two sums of four chips may be routed to the output of the Adder Tree in two clocks

Chip Descrambler Functional Description

- Read sixteen consecutive $T_c/2$ samples out of the Antenna Buffer corresponding to one finger, at the offset specified by the Path Searcher
- Mask out the unwanted half-chip samples
- Align the remaining eight samples (32 x 8 Barrel Shifter) with the Descrambling Code (Gold Code)
- Sign-extend the 8-bit data samples to 16 bits
- Multiply the eight aligned samples with the appropriate Despreading Codes

Chip Descrambler Functional Description

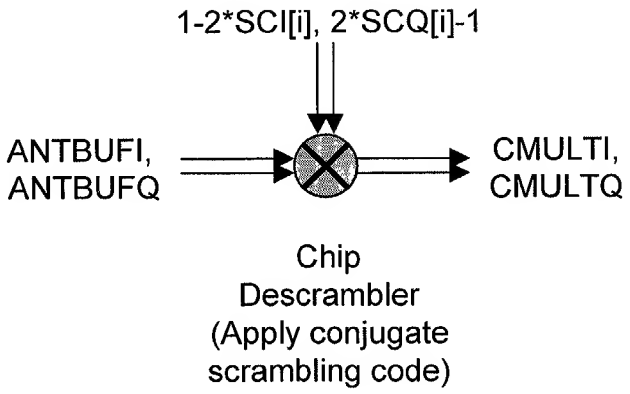


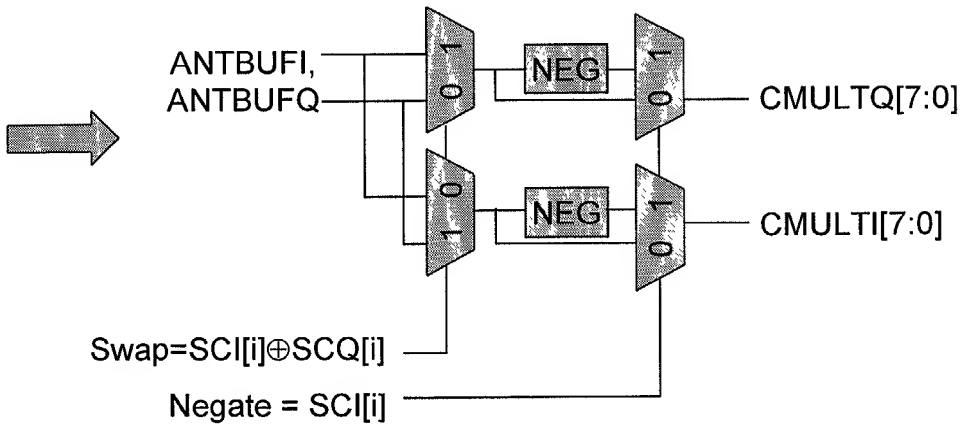
IN[31:16]=Q[15:0], IN[15:0]=I[15:0]

SCI	SCQ	OUT[31:16]	OUT[15:0]	SWAP/ CONJUGATE	NEGATE
0	0	-I	Q	Conjugate	
0	1	Q	I	Swap	
1	0	-Q	-I	Swap	Negate
1	1	I	-Q	Conjugate	Negate

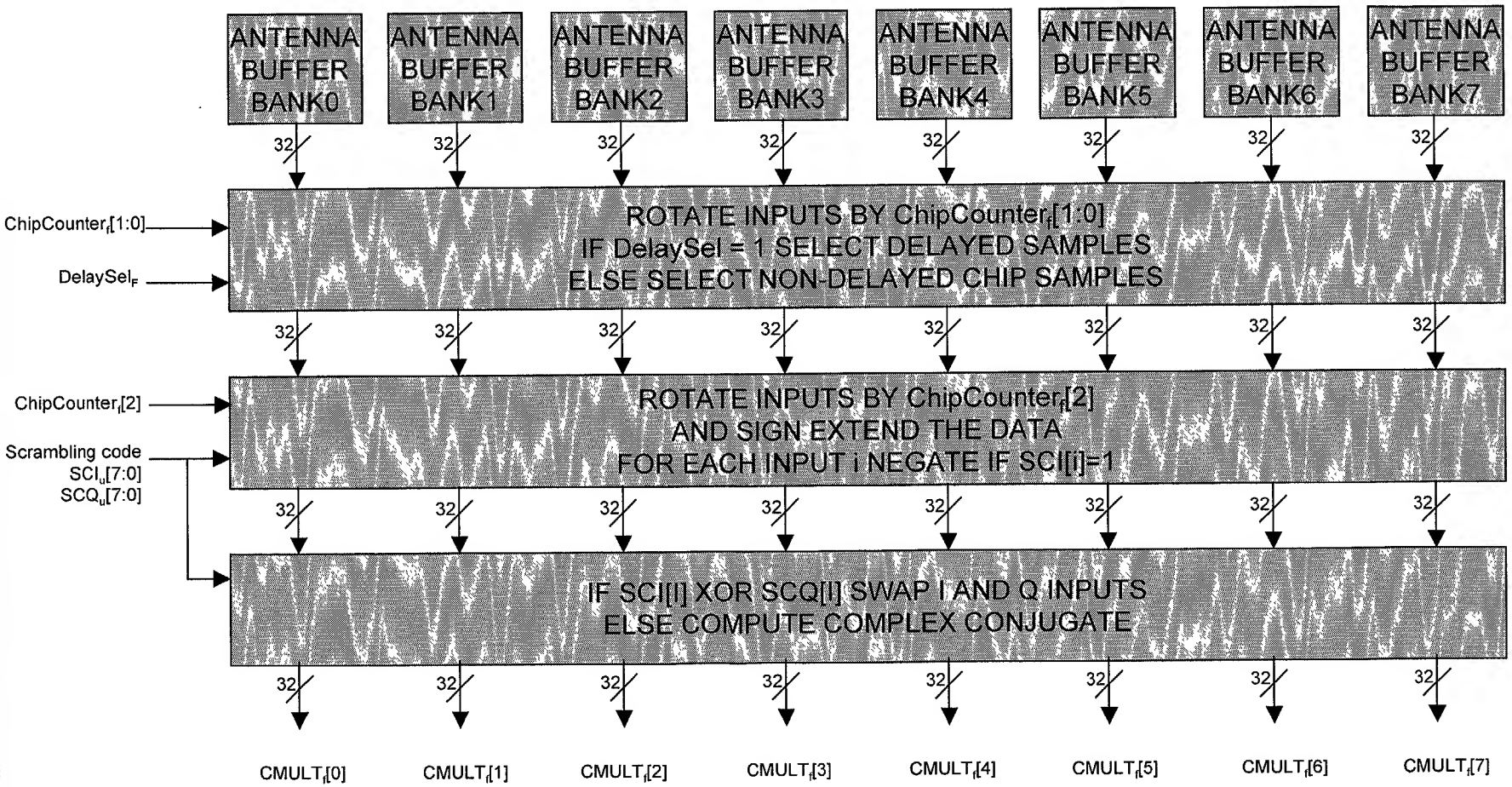
OUT[31:16]= CMULTI[15:0], OUT[15:0]=CMULTQ[15:0]

No multiplies or adds
required

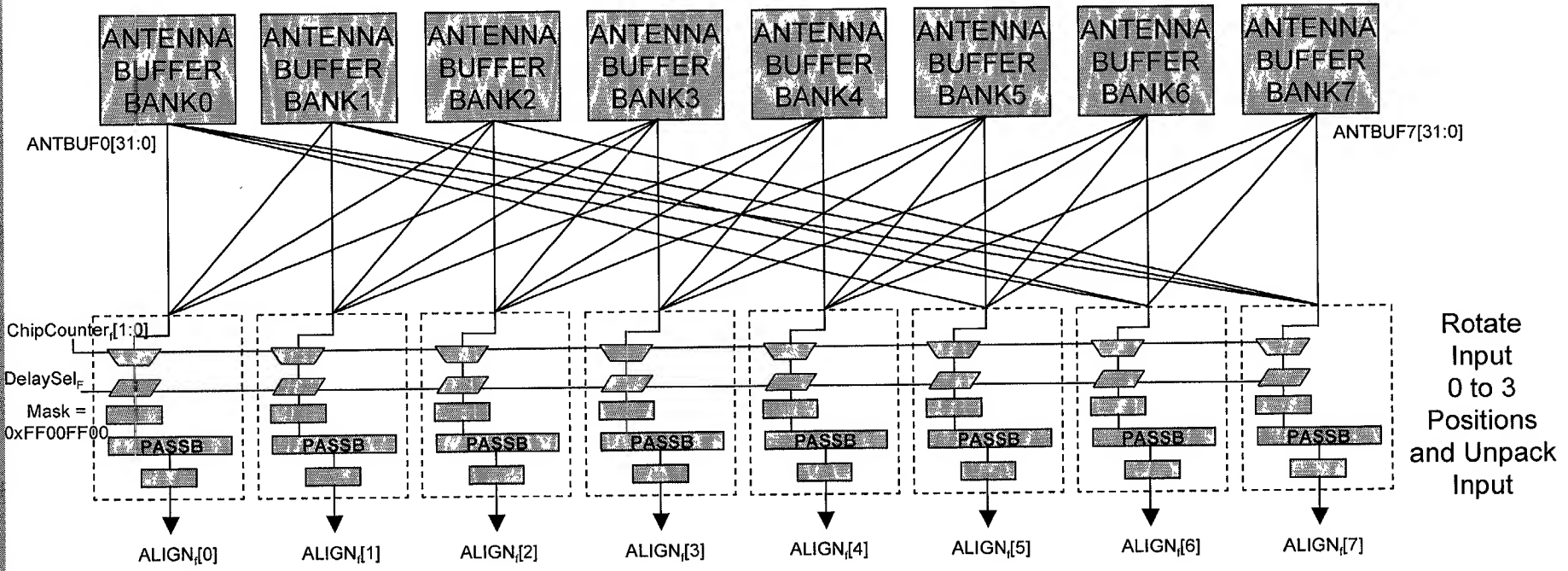




Chip Descrambler Block Diagram



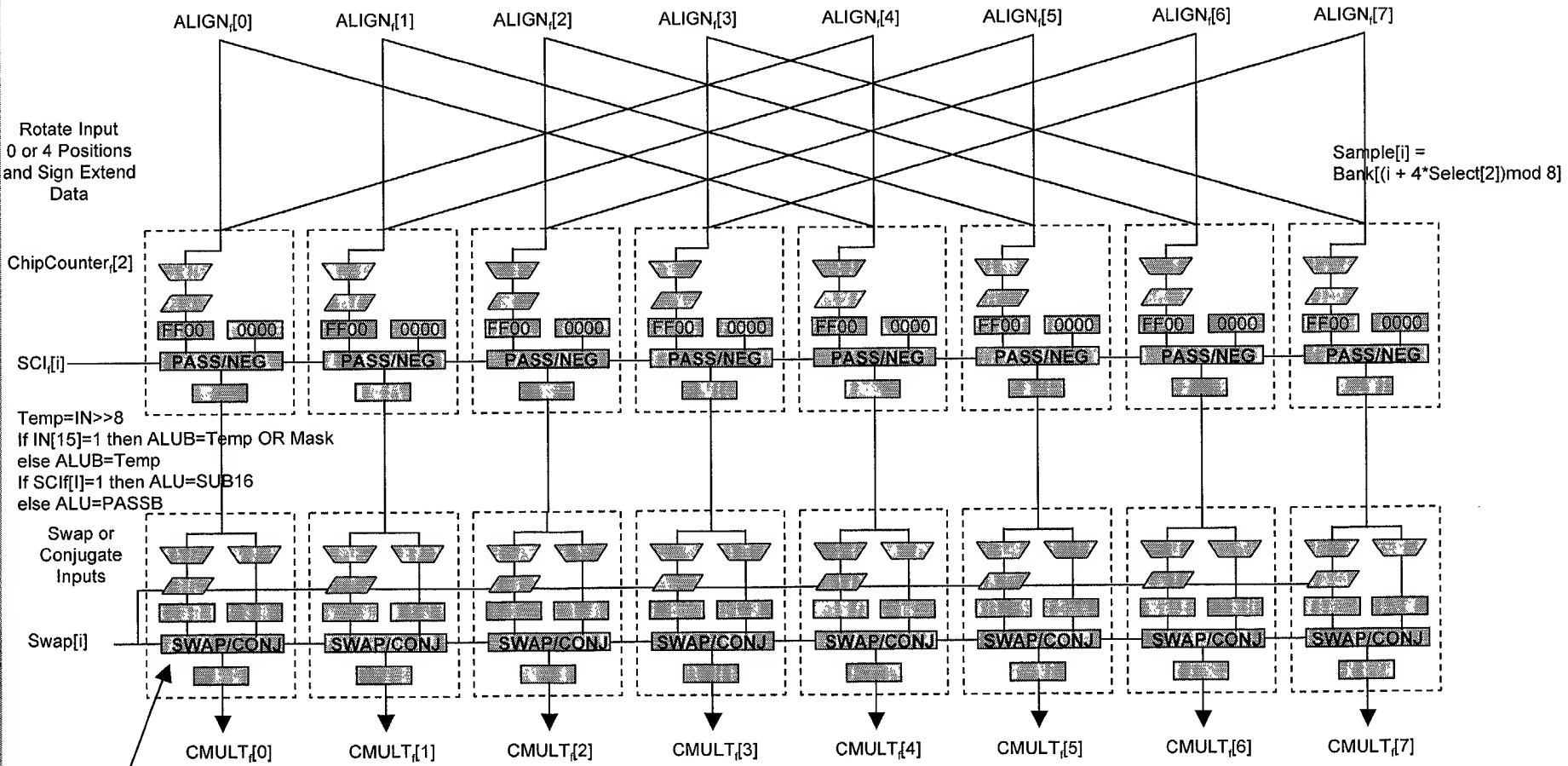
Chip Descrambler Input Alignment Implementation



Shifter Control
for Each Cell:
$DelaySel_F$=0: <<0
$DelaySel_F$ =1: <<8

Input Multiplexer Control:
$ALIGN_f[i] = Bank[(i + Select[1:0])mod\ 8]$

Chip Descrambler
Chip Multiplier Implementation



Swap Control for Each Cell:
If Swap[i] = 0
then ALUA[31:0]=IN[31:0],ALUB[31:0] =(IN[31:0]<<1 AND 0XFFFF)
ALU=SUB16
else ALUA[31:0]=IN[31:0],ALUB[15:0]=IN[31:16],ALUB[31:16]=IN[15:0]
ALU=PASSB

Output to Swap/Conjugate Unit

CHAMELEON SYSTEMS, INC.

Chip Descrambler Data Organization

Since the DPUs can only perform a SWAP or a Complex Conjugate, the I and Q inputs are assumed to be preswapped at the input to the Chip Descrambler circuit.

	SAMPLE[31:24]	SAMPLE[23:16]	SAMPLE[15:8]	SAMPLE[7:0]
SAMPLE BUFFER OUTPUTS: ANTBUF[31:0]	DQ_n[7:0]	DelayedDQ_n[7:0]	DI_n[7:0]	DelayedDI_n[7:0]
INPUT ALIGNMENT OUTPUTS: $ALIGN_F$[31:0]	DQ[7:0]	0x00	DI[7:0]	0x00
CHIP MULTIPLIER OUTPUTS: $CMULT_F$[31:0]	DI_n[15:0]	DI_n[7:0]	DQ_n[15:8]	DQ_n[7:0]
ADDER TREE OUTPUTS: ADD_F[31:0]	DI_n[15:0]	DI_n[7:0]	DQ_n[15:8]	DQ_n[7:0]

Where n is the sample n and n+ 1/2 is the next half-chip sample out of the Antenna Sample Buffer

Chip Descrambler Control Implementation

Control Input:
Scrambling Code Input $SCI_m[7:0]$, $SCQ_m[7:0]$
for User m

Data Inputs:
Eight Samples $DI_k[7:0]$
for k = 0 to 7

$SCI_n[i]$	$SCQ_n[i]$	$OUTI_m[i]$	$OutQ_m[i]$
0	0	$DQ_k[7:0]$	$-DI_k[7:0]$
0	1	$DI_k[7:0]$	$DQ_k[7:0]$
1	0	$-DI_k[7:0]$	$-DQ_k[7:0]$
1	1	$-DQ_k[7:0]$	$DI_k[7:0]$

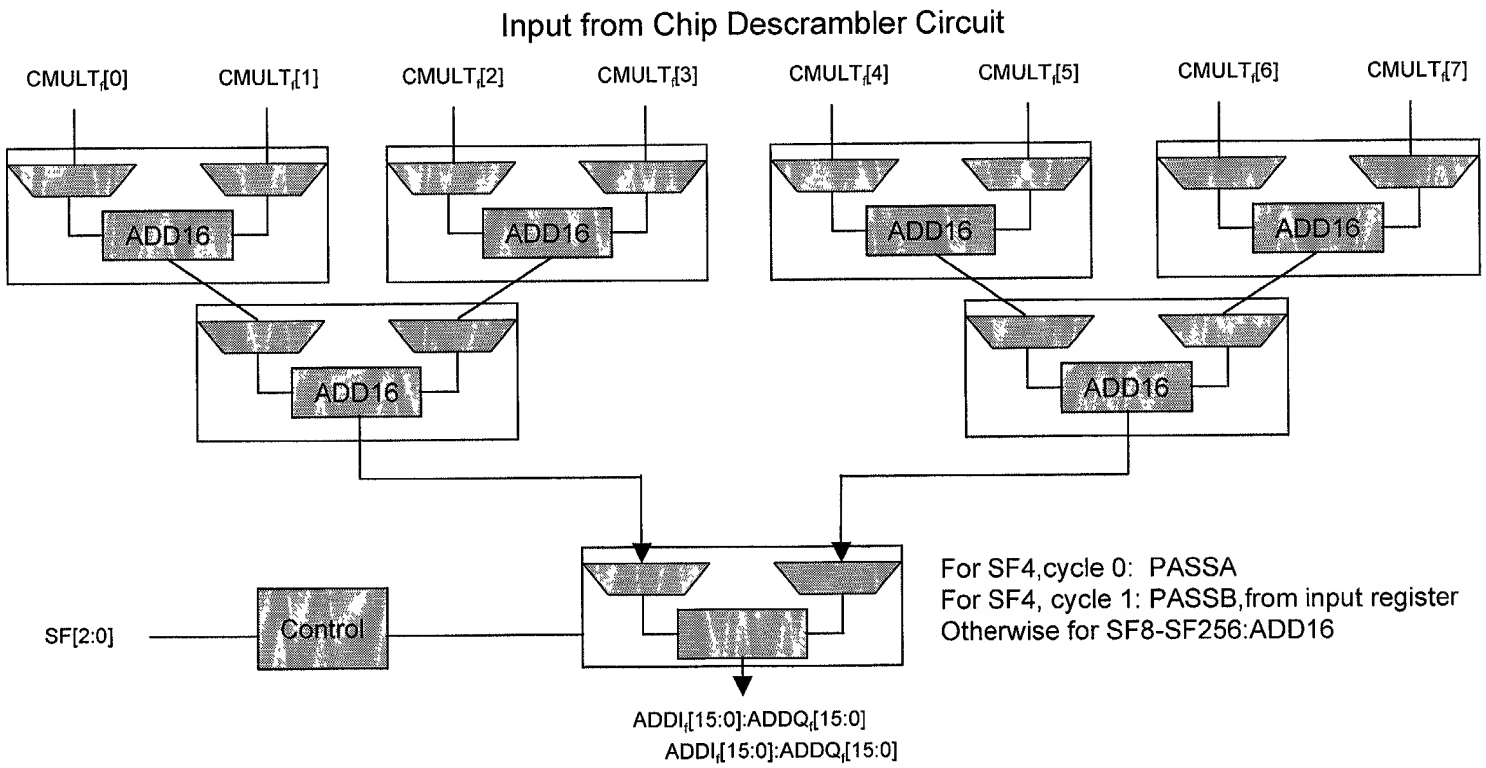
Chip Descrambler Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
    - 24 DPUs
    - 0 LSMs

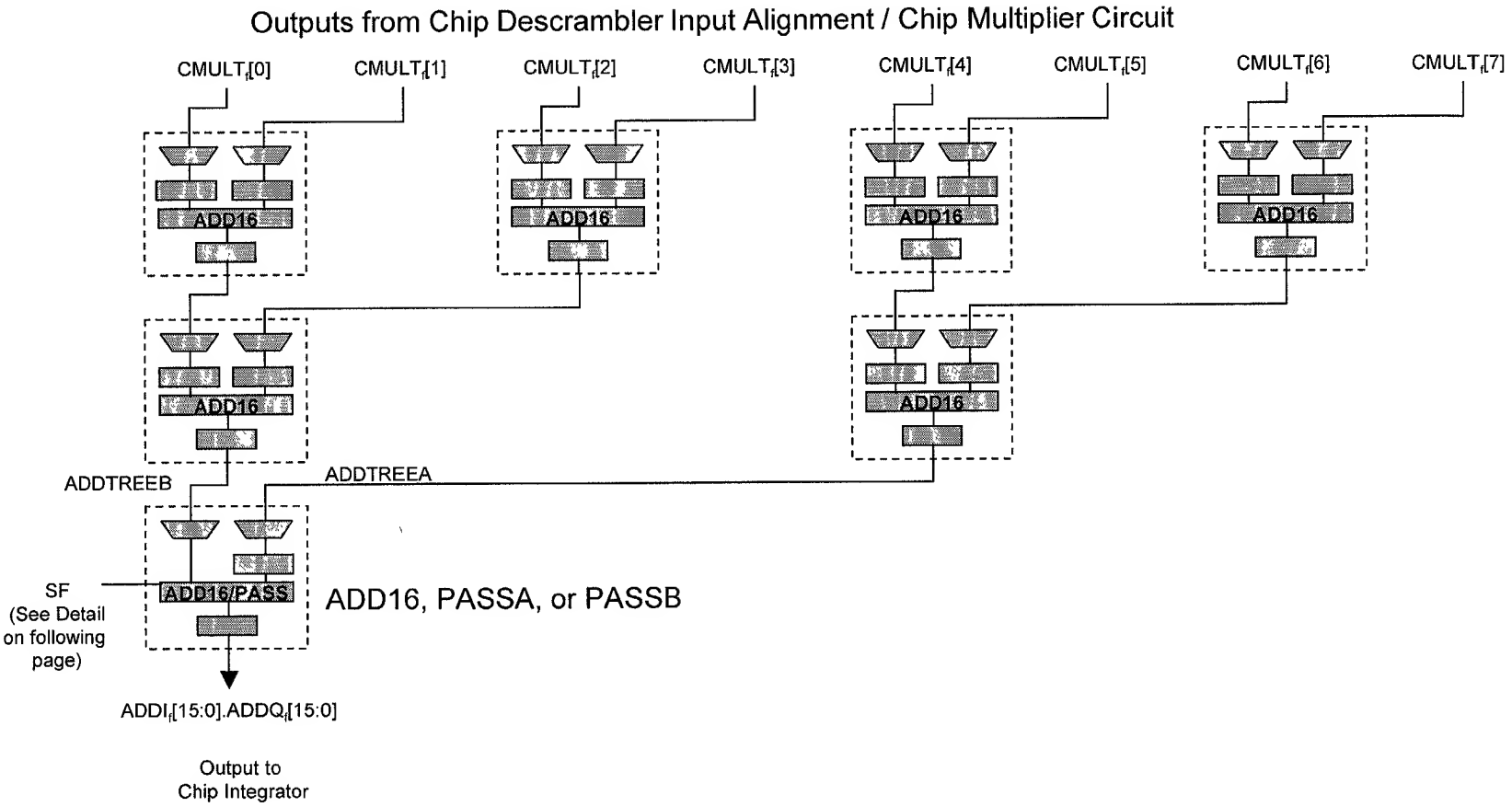
Chip Adder Tree Requirements and Assumptions

- Requirements
  - Add 8 chips per clock
  - Process 256 fingers @ 125 MHz
  - Process 512 fingers @ 250 MHz
- Assumptions
  - Inputs have been aligned on an even 8-chip boundary by the shifter in the Chip Descrambler circuit
  - All fingers with SF=4 will require 2 consecutive finger assignments so that two sums of four chips may be routed to the output of the Adder Tree in two clocks
  - Finger pairs allocated for SF=4 require that the first finger is assigned SF=4 and the second finger is assigned SF=0

Chip Adder Tree Functional Block Diagram

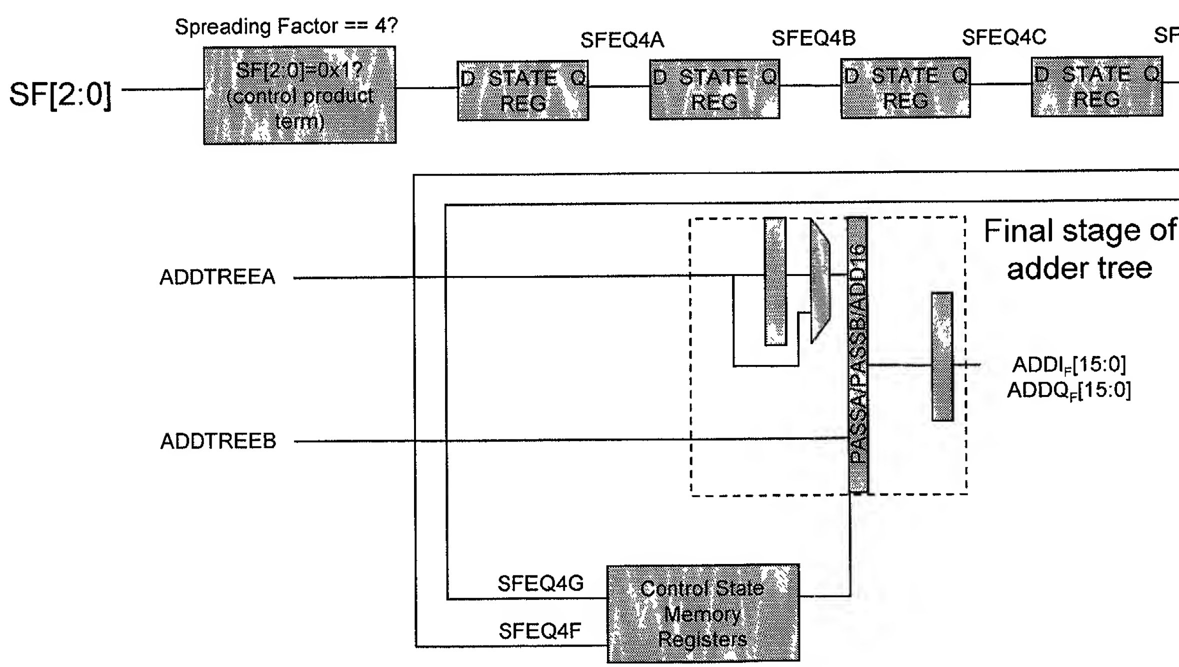


Chip Adder Tree Implementation



Chip Adder Tree
Control Implementation

We need delayed versions of the Spreading Factor (SF) to control the Adder Tree



This DPU is used to perform the final add in the 8:1 Chip Adder Tree. The control logic is looking for the SF=4 (SF[2:0]==0x1) condition. When the delayed decoding of the SF[2:0]==0x1 condition is detected, the DPU performs a PASSB operation. When SF=4 (SF[2:0]==0x1), we are guaranteed that the next cycle has SF=0 (SF[2:0]==0X0). The series of D-type flip-flops detect three possible states:

Control needs to generate three states for the DPU:
If (SFEQ4F:SFEQ4G == 10)
  ;Pass the first sum-of-four through the B arm of the DPU
  ALUB = ADDTREEB
  ALU = PASSB
Else if (SFEQ4F:SFEQ4G == 01)
  ;Pass the second sum-of-four through the B-arm of the DPU
  ;from the A-arm input register clocked the previous cycle
  ALUA = A-arm Register
  ALU = PASSA
Else
  ALUA = ADDTREEA
  ALUB = ADDTREEB
  ALU = ADD16

Chameleon Systems, Inc.

Chip Adder Tree Control Timing

Fn = Data for Finger n is valid

SF RAM Outputs	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19
UCB[2:0]=SF[2:0] valid in PLA	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
Descrambler ALIGN Output	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
Descrambler CMULT Output	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Adder Tree Final Stage Inputs	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
SFEQ4A=Reg. Decode of SF=4	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
SFEQ4B=Reg decode of SFEQ4A	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
SFEQ4C=Reg decode of SFEQ4B	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
SFEQ4D=Reg decode of SFEQ4C	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
SFEQ4E=Reg decode of SFEQ4D	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
SFEQ4F=Reg decode of SFEQ4E	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Final Stage DPU CSR Inputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Final Stage ALU Control Inputs	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
SFEQ4G=Reg decode of SFEQ4F	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11

Chip Adder Tree Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - 7 DPUs
  - 0 LSMs

Chip Integrator Requirements and Assumptions

- Requirements
  - Sum SF (Spreading Factor) partial sums into a single sum of SF chips
  - Prepare Data that is to be sent to the Channel Estimator in the ARC
- Assumptions
  - Average SF = 128
  - 256 chips input rate per 256 clocks @ 125 MHz

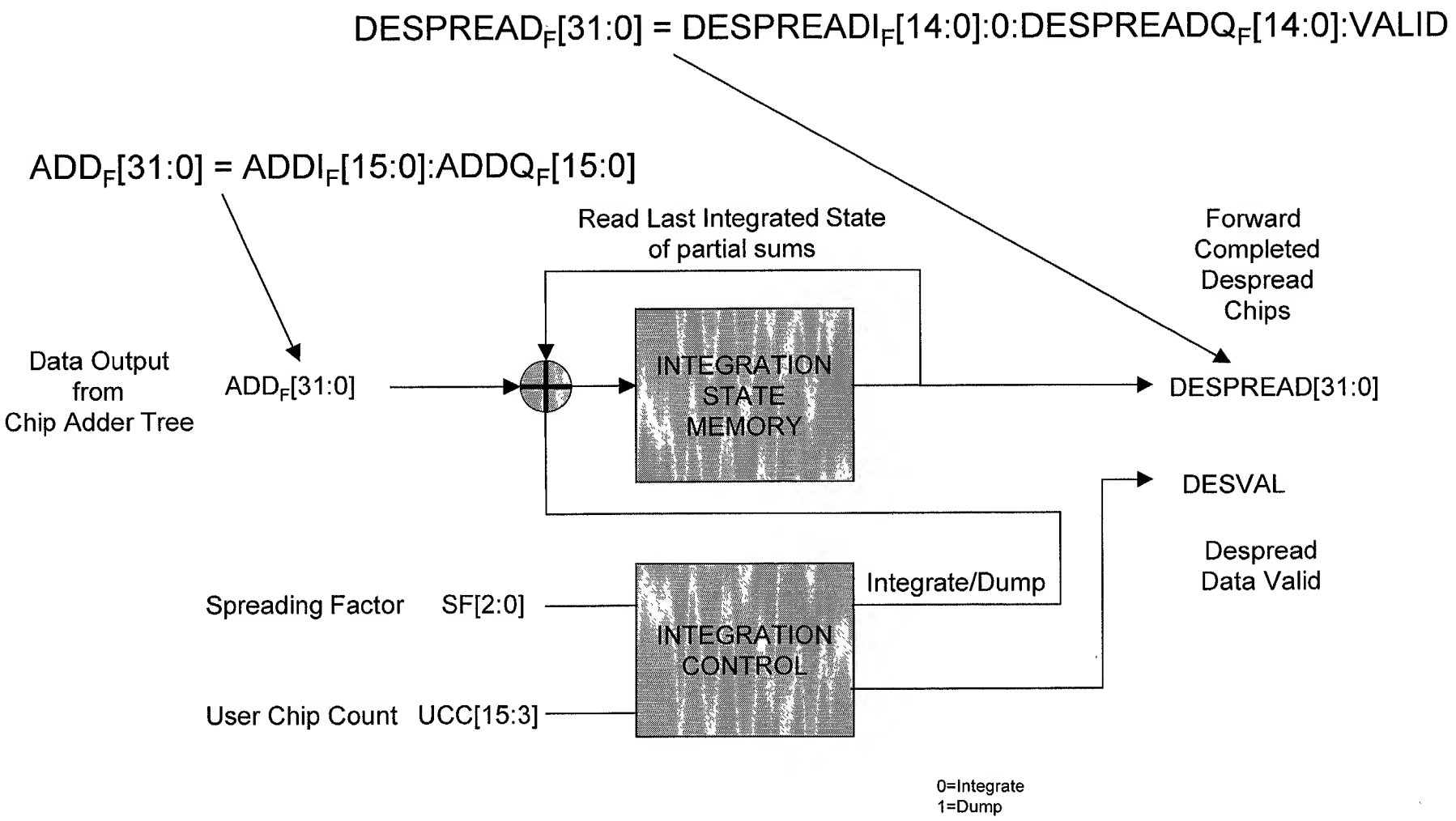
Chip Integrator Theory of Operation

- Read one complex sum from Chip Adder Tree every clock cycle
- The LSB position of the Q component of the accumulated sum will be used to hold a valid bit for the backend circuits
- The LSB of the sum with SF=4 will have this LSB "robbed" without a noticeable effect on the result
- The accumulated sums with SF=4-128 will have their full-precision results shifted up one bit position
- For data with SF=4-128, shift input data up one bit position, for data with SF=256, pass data straight through
- Input data has already been despread by 8 chips (4 chips for fingers with SF=4)
- Sum input data with partial sum until SF chips (SF/8 inputs) have been added together

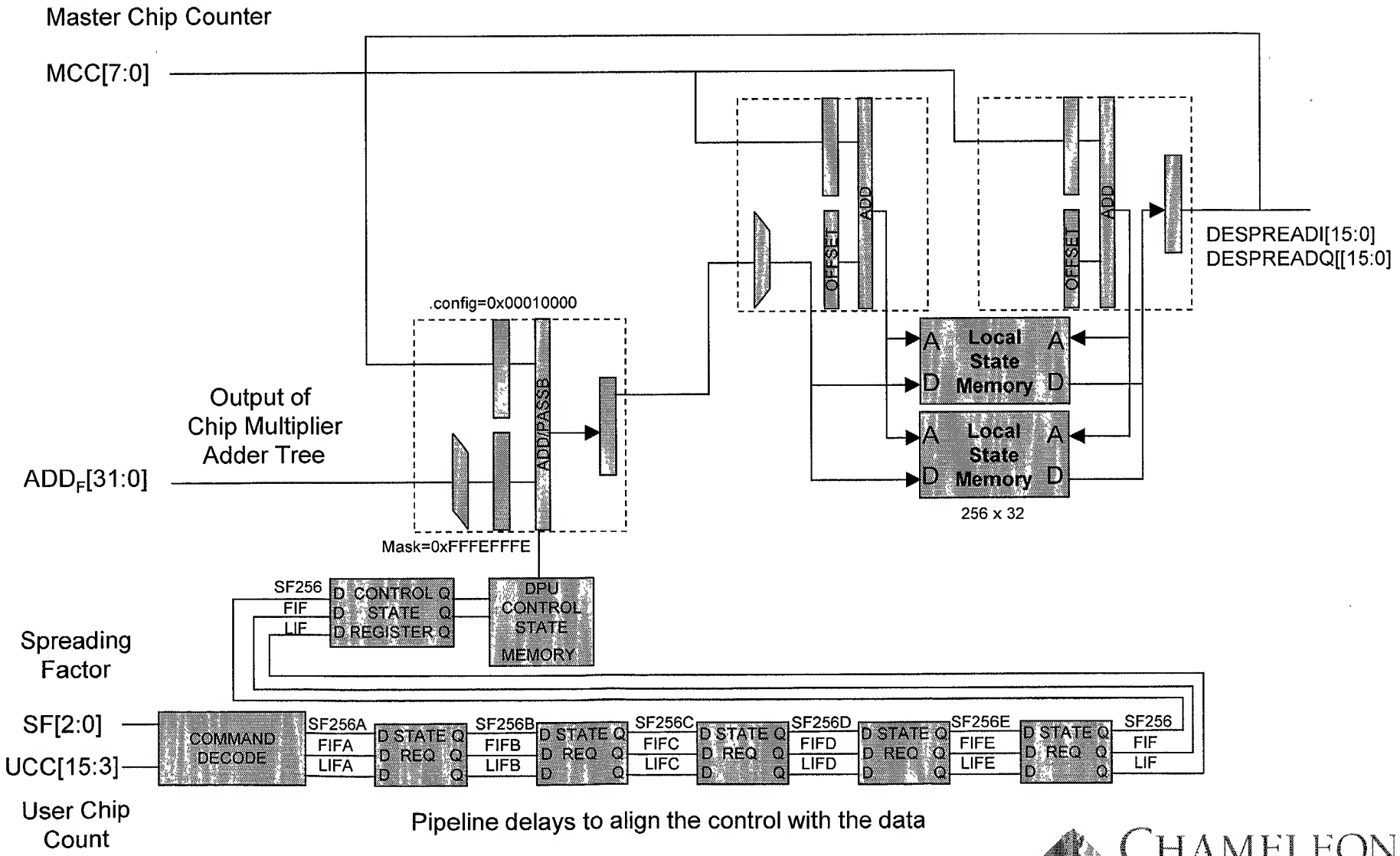
Chip Integrator Theory of Operation

- If the Chip Integrator input has SF=4-128 shift the input data up one bit position and set bits 0 and 16 to zero
- If the Chip Integrator input is the last 8-chip sample in a set of SF/8 chips, add it to the partial sum and place it into Chip Integrator Memory and set the valid bit (bit 0)
- If the Chip Integrator input is the first 8-chip sample in a set of SF/8 chips, place it into the Chip Integrator Memory and forward the previous despread sum

Chip Integrator Functional Block Diagram



Chip Integrator Datapath Implementation



Pipeline delays to align the control with the data

CHAMELEON SYSTEMS, INC.

Chip Integrator Control Implementation (1/2)

```
First In Frame FIF =   SF[2:0]==0                          ;Spreading Factor = 0
                     + SF[2:0]==1                          ;Spreading Factor = 4
                     + SF[2:0]==2                          ;Spreading Factor = 8
                     + (SF[2:0]==3 && UCC[3]== 0x0)        ;Spreading Factor = 16
                     + (SF[2:0]==4 && UCC[4:3]== 0x0)      ;Spreading Factor = 32
                     + (SF[2:0]==5 && UCC[5:3]== 0x0)      ;Spreading Factor = 64
                     + (SF[2:0]==6 && UCC[6:3]== 0x0)      ;Spreading Factor = 128
                     + (SF[2:0]==7 && UCC[7:3]==0x00)      ;Spreading Factor = 256

Last In Frame LIF =    SF[2:0]==0                          ;Spreading Factor = 0
                     + SF[2:0]==1                          ;Spreading Factor = 4
                     + SF[2:0]==2                          ;Spreading Factor = 8
                     + (SF[2:0]==3 && UCC[3]== 0x1)        ;Spreading Factor = 16
                     + (SF[2:0]==4 && UCC[4:3]== 0x3)      ;Spreading Factor = 32
                     + (SF[2:0]==5 && UCC[5:3]== 0x7)      ;Spreading Factor = 64
                     + (SF[2:0]==6 && UCC[6:3]== 0xF)      ;Spreading Factor = 128
                     + (SF[2:0]==7 && UCC[7:3]==0x1F)      ;Spreading Factor = 256

Spreading Factor Equals 256 SF256 = SF[2:0]==7             ;Spreading Factor = 256
```

Chip Integrator Control Implementation (1/2)

```
CASE (SF256:FIF:LIF)
  000:  ;Add 8-chip input to memory contents
        ALUA= AINPUT && Mask=0xFFFEFFFE
        ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
        ALU= ALUA + ALUB
  001:  ;Add 8-chip input to Chip Integrator Memory contents and set VALID bit
        ALUA= AINPUT && Mask=0xFFFEFFFE
        ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
        ALU= ALUA + ALUB + 1 ;Set VALID bit
  010:  ;Store 8-chip input into Chip Integrator Memory and forward previously despread sum
        ALUA= AINPUT && Mask=0xFFFEFFFE
        ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
        ALU= PASSB
  011:  ;Add 8-chip input to Chip Integrator Memory contents, set VALID bit,
        ;and forward previously despread sum
        ALUA= AINPUT && Mask=0xFFFEFFFE
        ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
        ALU= ALUA + ALUB + 1 ;Set VALID bit
```

Chip Integrator Control Implementation (2/2)

```
CASE (SF256:FIF:LIF) (CONTINUED)
 100:  ;Add 8-chip input to memory contents
       ALUA= AINPUT
       ALUB= BINPUT
       ALU= ALUA + ALUB
 101:  ;Add 8-chip input to Chip Integrator Memory contents and set VALID bit
       ALUA= AINPUT && Mask=0xFFFEFFFE
       ALUB= BINPUT &&& Mask=0xFFFEFFFE
       ALU= ALUA + ALUB + 1 ;Set VALID bit
 110:  ;Store 8-chip input into Chip Integrator Memory and forward previously despread sum
       ALUA= AINPUT
       ALUB= BINPUT
       ALU= PASSB
 111:  ;THIS INPUT COMBINATION IS NOT POSSIBLE
       ;ARBITRARILY ASSIGN THE SAME COMMAND AS CASE 110
       ALUA= AINPUT
       ALUB= BINPUT
       ALU= PASSB
```

Chip Integrator Control Timing

Fn = Data for Finger n is valid

SF/UCC RAM Outputs	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19
SF/UCC valid in PLA	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
Descrambler ALIGN Output	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
Descrambler CMULT Output	F254	U62A	U62B	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Adder Tree Final Stage Outputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
INTA=Reg Decode of SF/UCC	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
INTB= Reg decode of INTA	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
INTC= Reg decode of INTB	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
INTD= Reg decode of INTC	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
INTE= Reg decode of INTD	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
DataValid CSR Inputs	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
DataValid ALU Inputs	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
DataValid Output Register Valid	F249	F250	F251	F252	F253	F254	F255	F0	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
Integrate/Dump Output Reg Val	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Integrate/Dump CSR Inputs	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
Integrate/Dump ALU Inputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12

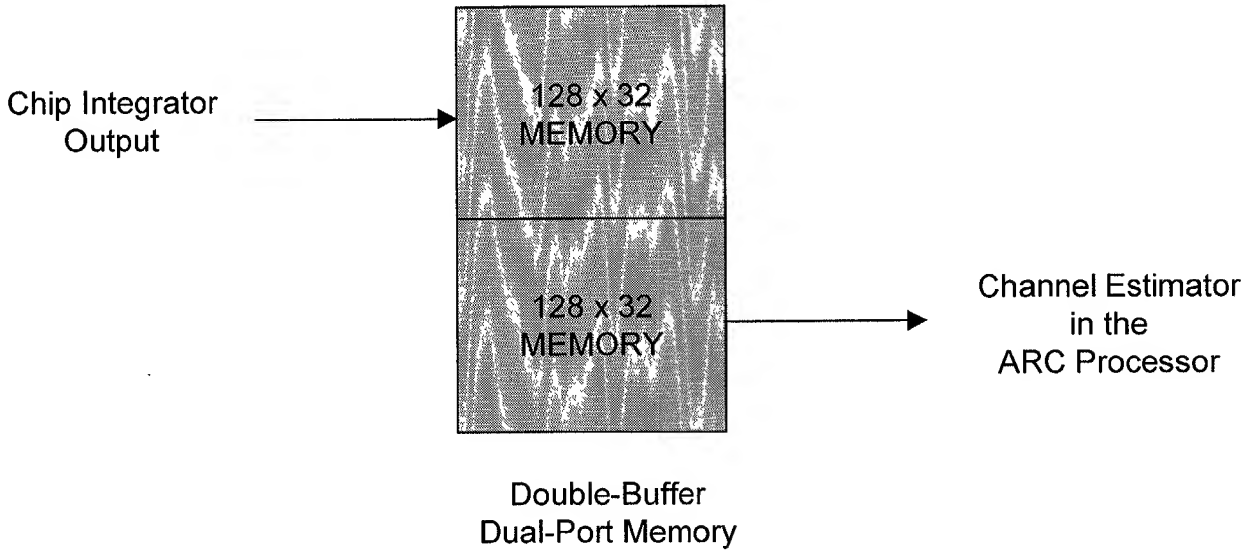
Chip Integrator Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - 3 DPUs
  - 2 LSMs

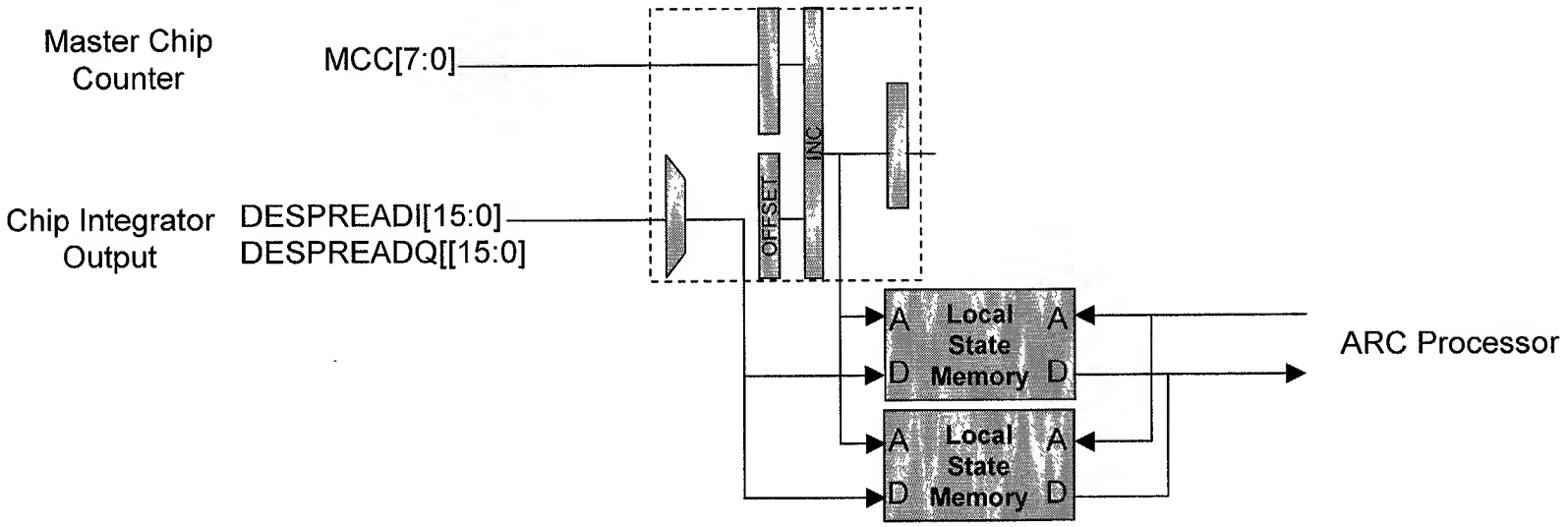
Channel Estimator Data Input Buffer Requirements and Assumptions

- Requirements
  - Provide a path between the Chip Integrator in the Chameleon fabric and the ARC processor
  - Double-buffer 128 despread pilot symbols every 256 T_c
- Assumptions
  - Input written by Chip Accumulator
  - Output is read by the ARC core
  - All Control Channel data has SF=256

Channel Estimator Data Input Buffer Functional Block Diagram



Channel Estimator Data Input Buffer Implementation



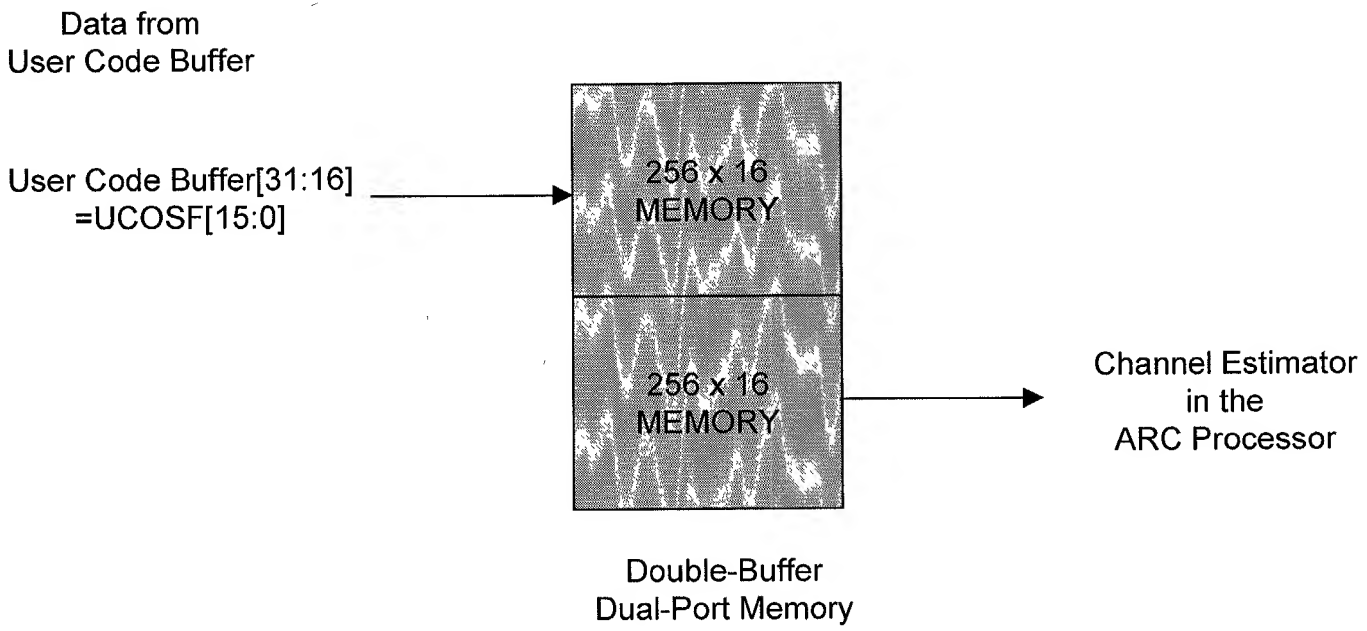
Channel Estimator Data Input Buffer Resource Requirements

- 32 User Implementation @ 125 MHz
  - 1 DPU
  - 2 LSMs
- 48 User Implementation @ 187.5 MHz
  - 1 DPUs
  - 4 LSMs
- 64 User Implementation @ 250 MHz
  - 1 DPUs
  - 6 LSMs

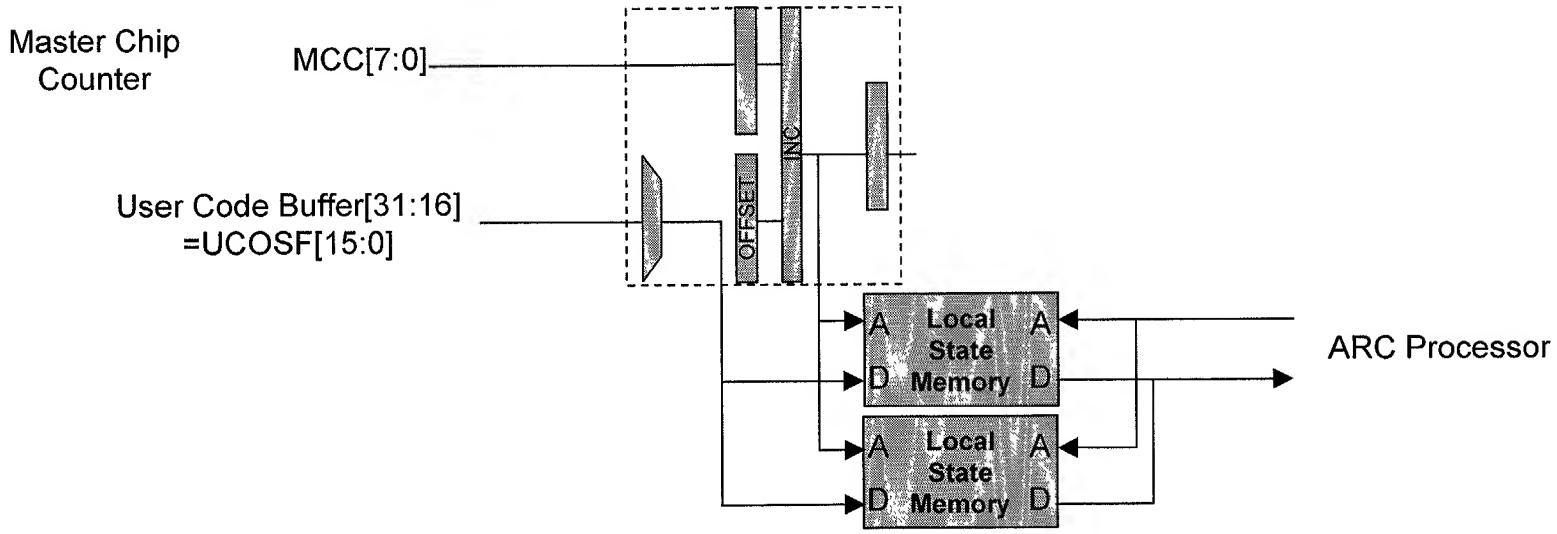
Channel Estimator UCC Input Buffer Requirements and Assumptions

- Requirements
  - Provide a buffer containing the present User Chip Counter (UCC) value for each of the 128 Pilot chips
  - Provide a buffer containing the present User Chip Counter (UCC) value for each of the 128 Data chips
  - Double-buffer 256 UCCs every 256 T_c
- Assumptions
  - Input written by User Code Buffer
  - Output is read by the ARC core

Channel Estimator UCC Input Buffer Functional Block Diagram



Channel Estimator UCC Input Buffer Implementation



Channel Estimator UCC Input Buffer Resource Requirements

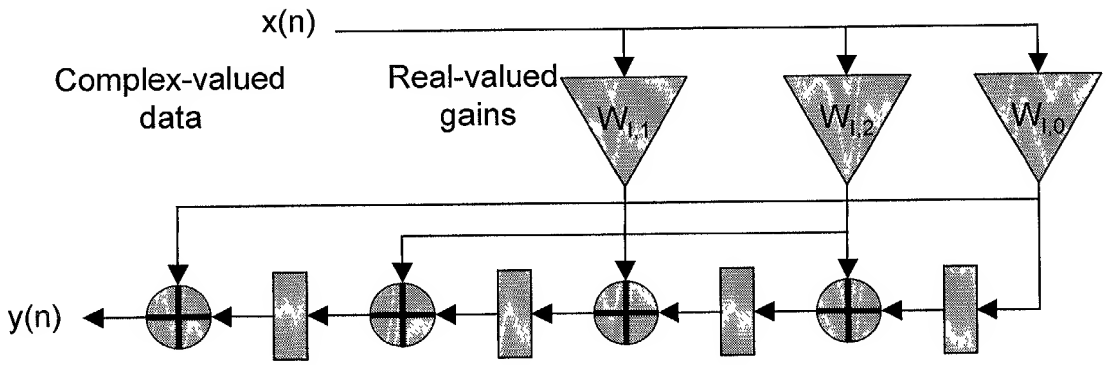
- 32 User Implementation @ 125 MHz
  - 1 DPU
  - 2 LSMs
- 48 User Implementation @ 187.5 MHz
  - 1 DPUs
  - 4 LSMs
- 64 User Implementation @ 250 MHz
  - 1 DPUs
  - 6 LSMs

Channel Estimator Requirements and Assumptions

- Requirements
  - Compensate the user data given the characteristics of the Pilot Channel data using XXX filtering
- Assumptions
  - All channel estimation is performed in the ARC processor
  - The Channel estimation is performed after the Pilot Channel has been despread (SF=256) and is significantly slower than the chip rate

FIR Filters for Channel Estimation

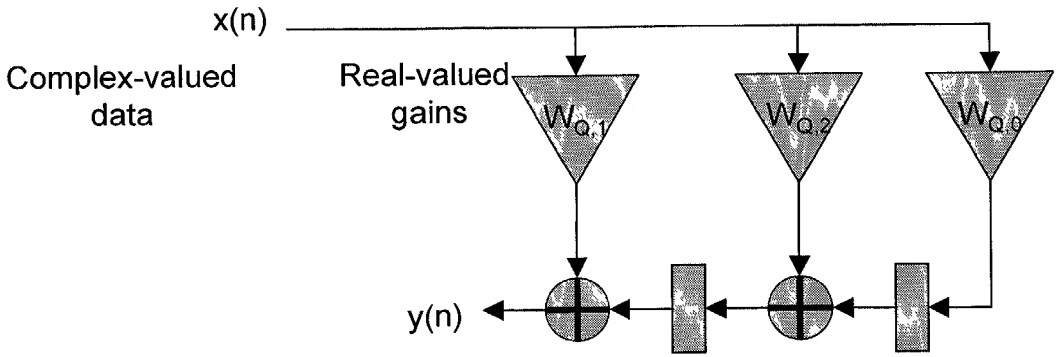
5-Tap Symmetric FIR Filter



Required Ops/Sample:
- 6 Multiplies
- 4 Additions
- 3 Post-Multiply Packing Ops

Allow Multiple Clocks per Sample:
- 2 MUL for multiplies
- 1 DPU for additions
- 1 DPU for packing

3-Tap Non-symmetric FIR Filter



Required Ops/Sample:
- 6 Multiplies
- 2 Additions
- 3 Post-Multiply Packing Ops

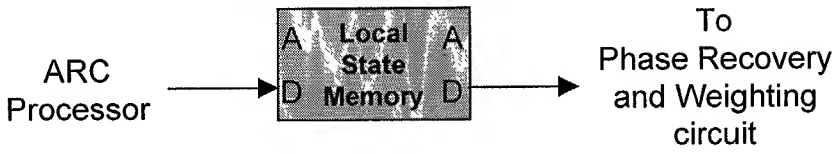
Allow Multiple Clocks per Sample:
- 2 MUL for multiplies
- 1 DPU for additions
- 1 DPU for packing

CHAMELEON SYSTEMS, INC.

Channel Estimator Output Buffer Requirements and Assumptions

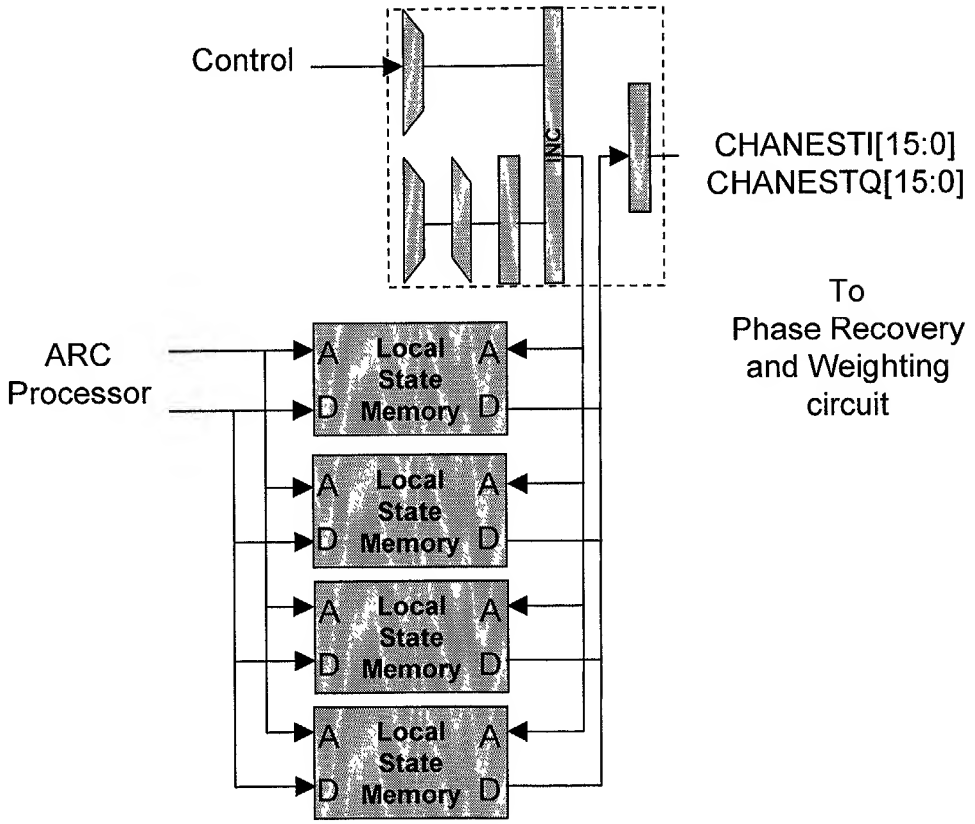
- Requirements
  - Provide a buffer between the Channel Estimation Filter and the Phase Recovery circuits
  - Provide a double buffer memory to prevent race conditions
- Assumptions
  - One 32-bit (16-bit I, 16-bit Q) Channel Compensation Weight word per pilot is required
  - 32 Users required
  - Must provide double buffer for proper operation
  - Channel Compensation word is sample and held on one time-slot (2560 T_c) boundaries

Channel Estimator Output Buffer Functional Block Diagram



256 Fingers: 256 fingers x 32-bit words x 2 banks (ping, pong)

Channel Estimator Output Buffer Implementation



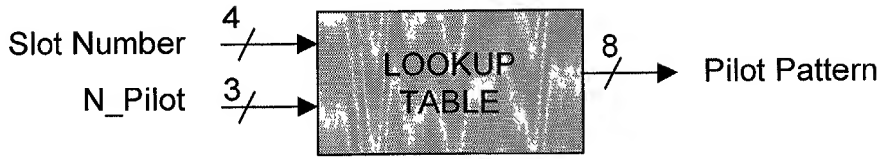
Channel Estimator Output Buffer Resource Requirements

- 32 Users Implementation
  - 1 DPU
  - 2 LSMs
- 48 Users Implementation
  - 2 DPU
  - 3 LSMs
- 64 Users Implementation
  - 2 DPU
  - 4 LSMs

Pilot Pattern Generator Requirements and Assumptions

- Requirements
  - Provide simple lookup-table approach
- Assumptions
  - Physically resides in ARC processor
  - N_Pilot = 3,4,5,6,7, or 8
  - The pilot is a function of slot
  - A single LSM is sufficient to contain the lookup-table

Pilot Pattern Generator Functional Block Diagram



Pilot Pattern Generator Memory Map Address Bit Definitions

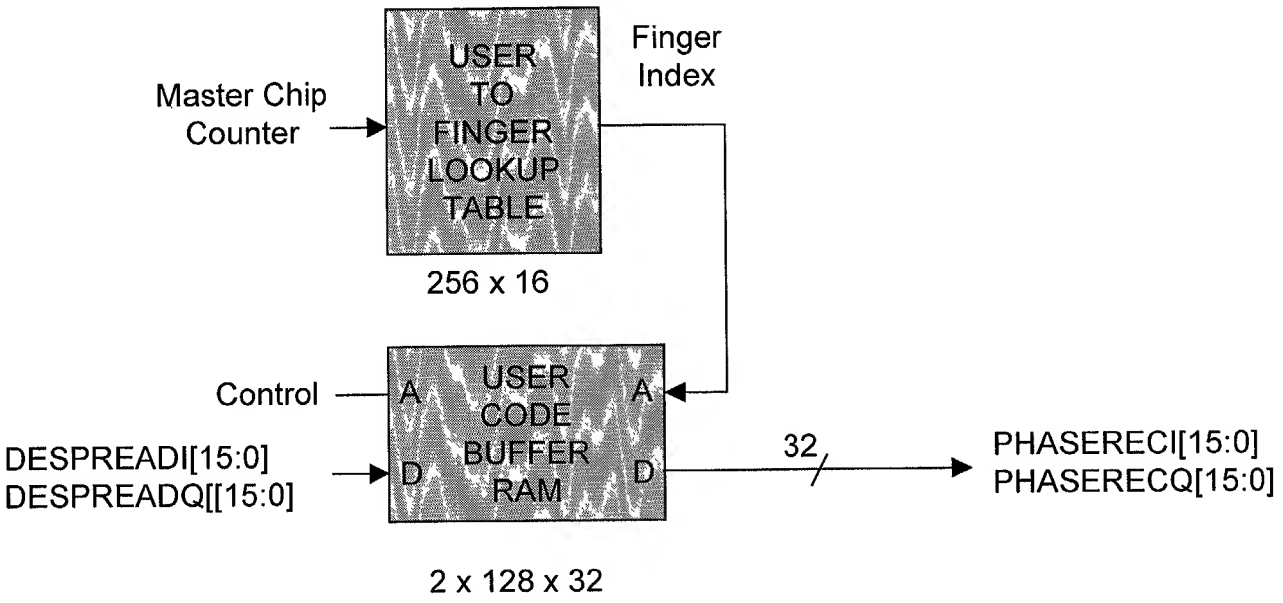
N_Pilot	PilotGenAddr[6:4]
3	0
4	1
5	2
6	3
7	4
8	5

SlotNumber	PilotGenAddr[3:0]
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14

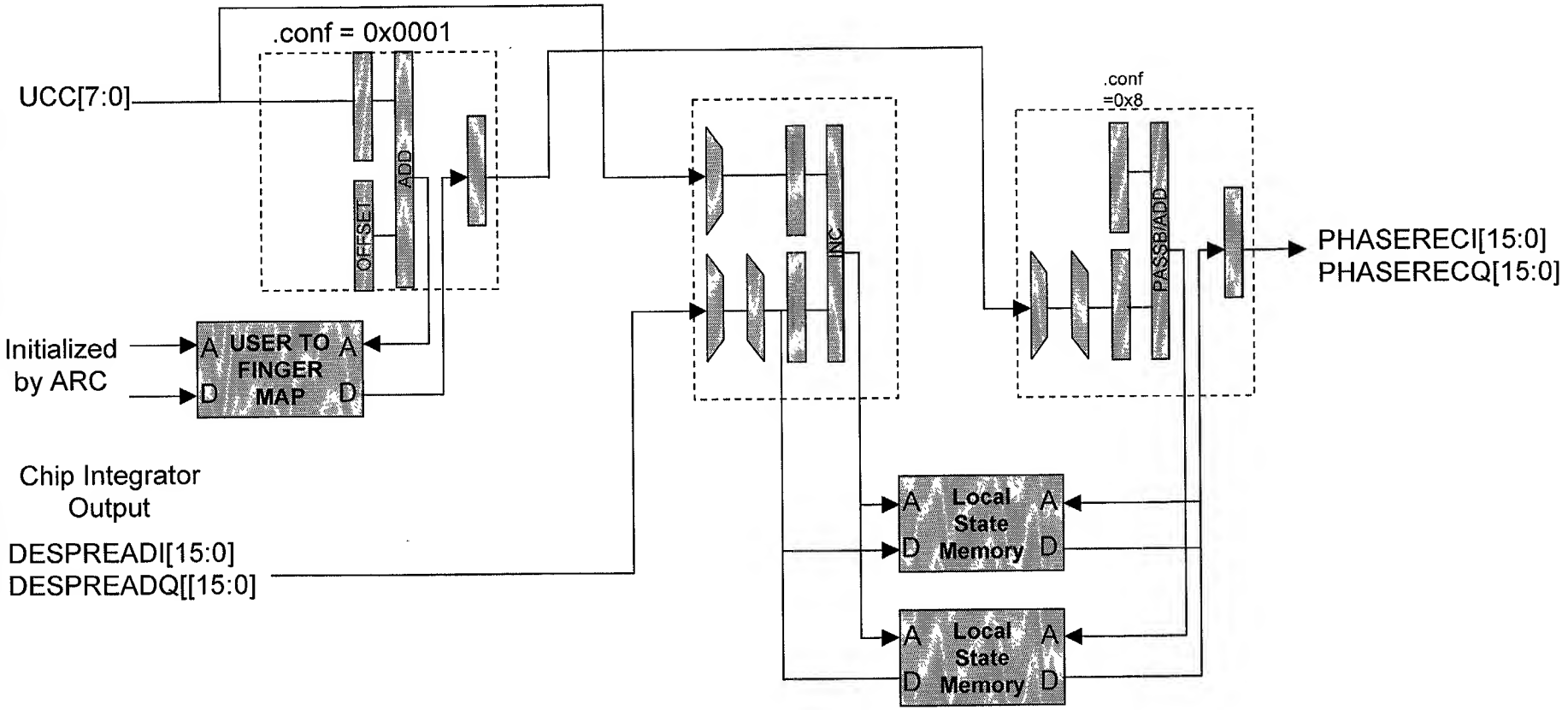
Phase Recovery Input Buffer Requirements and Assumptions

- Requirements
  - Provide a double buffer between the Chip Integrator circuit and the Phase Recovery circuit
  - Buffer 128 fingers every 256 clocks
- Assumptions
  - Only the delayed data channels are buffered
  - Pilot Channel data is not buffered

Phase Recovery Input Buffer Block Diagram



Phase Recovery Input Buffer Implementation



CHAMELEON SYSTEMS, INC.

Phase Recovery and Weighting Requirements and Assumptions

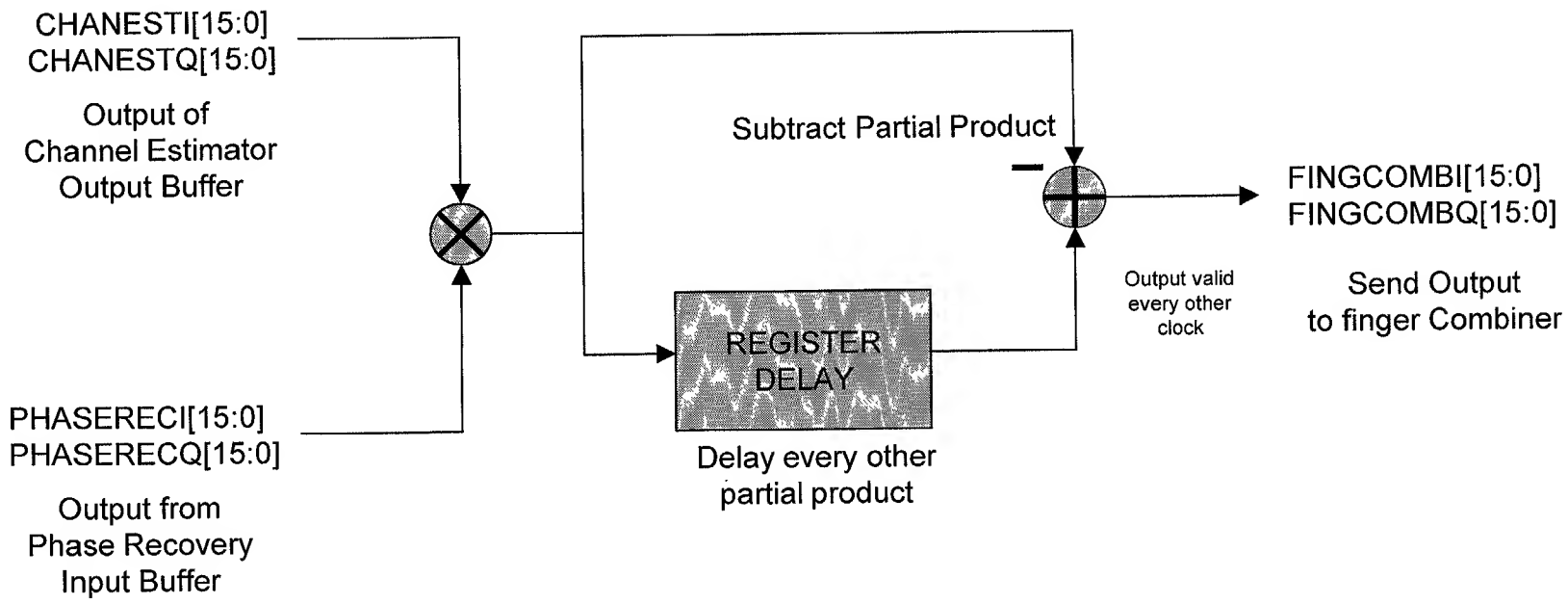
- Requirements
- Assumptions
  - A complex multiplication of (A +jB) * (C +jD)

    = (AC - BD) +j(AD + BC)
  - We are only interested in the real part of the output:

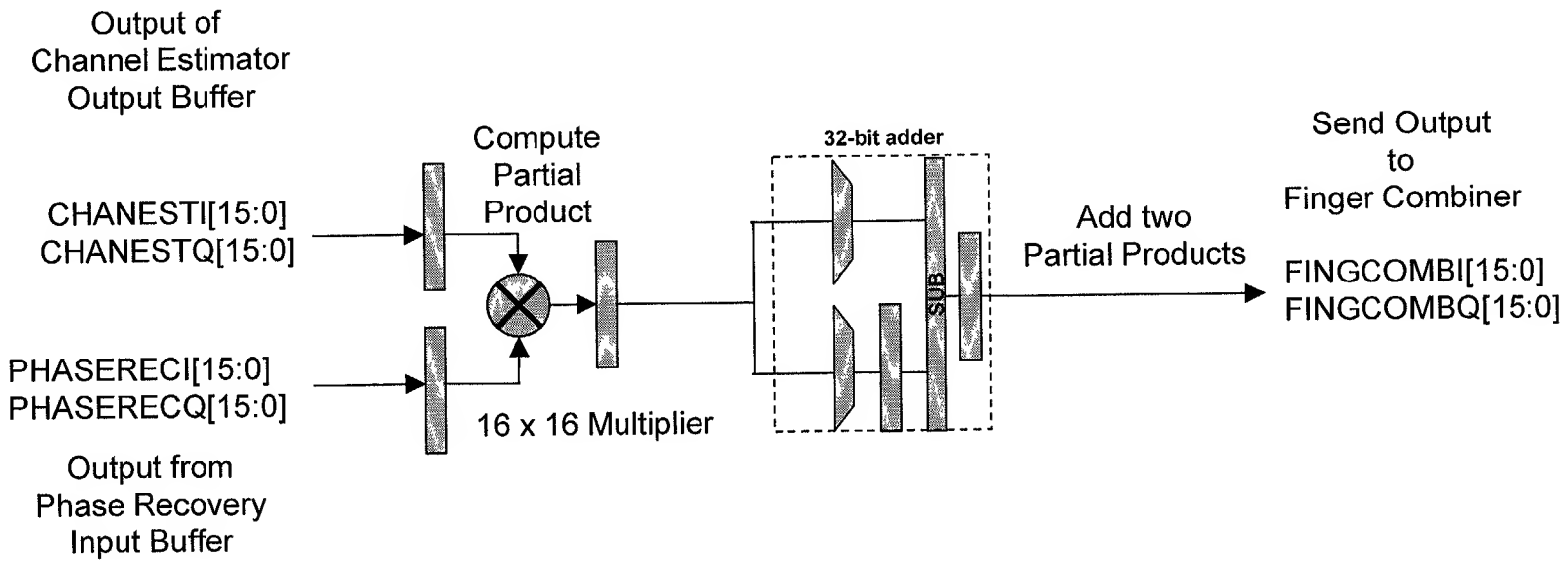
    Re{(A +jB) * (C +jD)} = AC - BD

  We have extra cycles so only one multiplier is required

Phase Recovery and Weighting Functional Block Diagram



Phase Recovery and Weighting Implementation



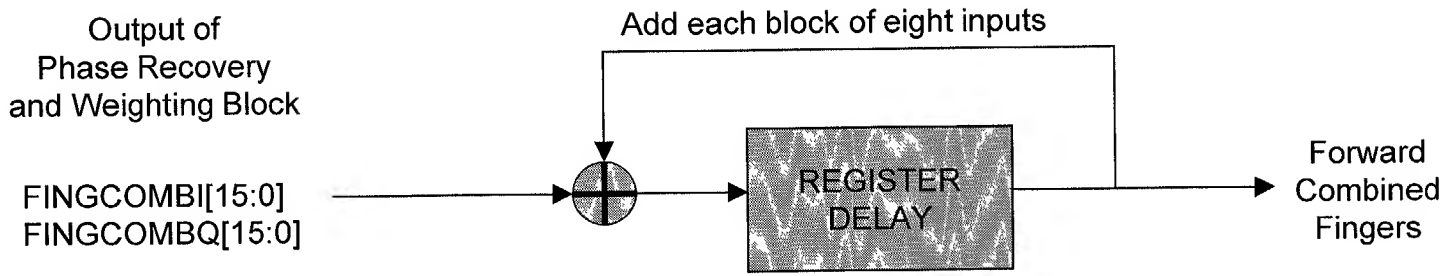
Phase Recovery and Weighting Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - 1 DPU
  - 1 Multiplier

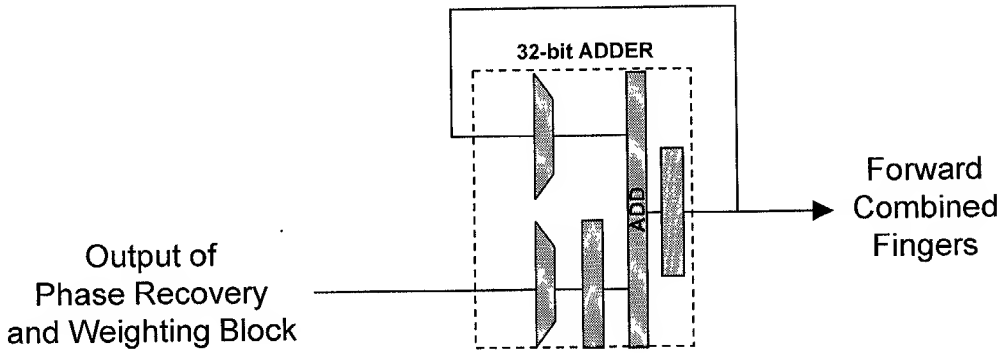
Finger Combiner Requirements and Assumptions

- Requirements
  - Must be able to combine up to six fingers
- Assumptions
  - Extra cycles can be wasted as long as the TPCG < 300us
  - Allocate timing such that the circuit is always adding 8 fingers
  - The ARC processor may assign up to 8 fingers to each user
  - The ARC processor assigns zeroes to the unused fingers in an eight finger block

Finger Combiner Functional Block Diagram



Finger Combiner Implementation



Finger Combiner Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
    - 1 DPU

General Timing and Control Finger Tracking within the Rake Receiver

- Upon initial acquisition, the Rake Receiver is tasked such that the Scrambling Code is set to an offset that places the (up to) six fingers in the Antenna Sample Buffer window such that zero path delay corresponds to the beginning of the buffer window
- As the mobile user moves toward or away from the base station (Node B), the fingers will move within the Antenna Sample Buffer window as tasked by the Path Searcher

Spreading Factor Mapping Assignments

- The following table lists the values corresponding to the various Spreading Factors (SF):
- Note that users requiring SF=4 must assign each finger to two successive finger resources per finger according to the following rules
  - At finger resource n assign SF=4
  - At finger resource n+1 assign SF=0 (used to denote second part of SF=4 finger)

SPREADING FACTOR	SF MEMORY CONTENTS
0	0
4	1
8	2
16	3
32	4
64	5
128	6
256	7

Chameleon Systems Rake Receiver CS2112 Device Utilization for 32 Users

KERNEL	DATA PATH UNITS (DPUs)	LOCAL STATE MEMORIES (LSMs)	MULTIPLIERS
Gold Code Generator	14	8	2
Channel Code Generator / Multiplier	4	2	0
Antenna Sample Buffer	22	26	0
Chip Descrambler	24	0	0
Chip Adder Tree	7	0	0
Chip Integrator	3	2	0
Channel Estimator Data Input Buffer	1	2	0
Channel Estimator UCC Input Buffer	1	2	0
Channel Estimator Output Buffer	1	2	0
Phase Recovery Input Buffer	3	3	0
Phase Recovery and Weighting	1	0	1
Finger Combiner	1	0	0
Master Chip Counter	2	0	0
TOTALS:	84	47	3
TOTAL AVAILABLE	84	48	24
PERCENT UTILIZED	100%	98%	13%

```
/*
 $Id: galois.c,v 1.4 2000/03/21 14:49:27 rollins Exp $

 galois.c

 (c) 2000 Chameleon Systems Inc.
     Algorithms in Reconfigurable Silicon

 Mark Rollins
 14-Mar-2000
*/

#include "galois.h"

/*
 ------------------------------------------------------------
 Polynomial Multiplication Modulo f(x) for GF(2^25)

 This version is hard coded for GS(2^25).

 Since we need to multiply polynomials of order 24, the result
 will not fit in a 32-bit register. We need to manage two
 such registers (upper and lower).

 Polynomial multiplication is just simply shifts and adds
 with the added complexity of managing the two registers.

 Polynomial reduction modulo f(x) is performed by adding in
 shifted correction terms f(x) which line up with the undesired
 higher order 1's in the product (lying in the 25-th bit and above).
 ------------------------------------------------------------
*/

int poly_mult_modulo_fx_2p25( int a_x, int b_x, int f_x )
{
    int lower, upper;
    int shft_a, shft_b, upper_b;
    int fsl, fsu;
    int i;

    /* Initialize */
    upper = 0;
    upper_b = 0;
    shft_a = a_x;
    shft_b = b_x;
    lower = ((shft_a & 1) == 1) ? b_x : 0; /* Shift 0 */

    /* Shifts 1 to 7 remain in lower register */
    for (i=1; i <= 7; i++) {
      shft_a >>= 1;
      shft_b <<= 1;
      if ((shft_a & 1) == 1)
          lower = lower ^ shft_b;
    }

    /* Shifts 8 to 24 spread across lower & upper registers */
    for (i=8; i <= 24; i++) {
```

```
      shft_a >>= 1;
      upper_b <<= 1;
      upper_b += ( (shft_b&0x80000000) >> 31 );
      shft_b <<= 1;
      if ((shft_a & 1) == 1) {
          lower = lower ^ shft_b;
          upper = upper ^ upper_b;
      }
    }

    /*
       Perform modulo f(x) reduction on high order bits:
       - check if bits 48, 47, ..., 25 are unity
       - if yes, add shifted versions of f(x)
    */

    /* Bits 48 to 32 spread across lower & upper registers */
    fsu = f_x >> 9;
    for (i=16; i>=0; i--) {
      fsl = f_x << (7+i);
      if ( ((upper>>i)&1) == 1 ) {
          upper = upper ^ fsu;
          lower = lower ^ fsl;
      }
      fsu >>= 1;
    }

    /* Bits 31 to 25 remain in lower register */
    for (i=31; i >= 25; i--) {
      fsl = f_x << (i-25);
      if ( ((lower>>i)&1) == 1 )
          lower = lower ^ fsl;
    }
/*     printf("Upper: %x\n", upper); */
/*     printf("Lower: %x\n", lower); */

    return lower;
}

/*
 ---------------------------------------------------------------
 Polynomial Division With Max Degree 25

 Determine a(x) = ( g(x) / f(x) )_{deg<25}

 where
          f(x) is a primitive polynomial of degree '25'
        g(x) is a polynomial of degree < '25'
        a(x) is a polynomial of degree < '25'

 The higher order terms of a(x) are not calculated.
 ---------------------------------------------------------------
*/

int poly_divide_max_degree_25( int g_x, int f_x )
{
    int i, a_x, shft_g;
```

```
    a_x = 0;
    shft_g = g_x;
    for (i=0; i < 25; i++) {
      if ( (shft_g & 1) == 1 ) {
          shft_g = shft_g ^ f_x;
          a_x += (1 << i);
      }
      shft_g >>= 1;
    }
    return a_x;
}

/*
 ------------------------------------------------------------
 Polynomial Multiplication With Max Degree 25
 Determine
            g(x) = ( f(x)b(x) )_{deg<25}
 where
            f(x) is a primitive polynomial of degree '25'
            b(x) is a polynomial of degree < '25'
            g(x) is restricted to have degree < '25'

 The higher order terms of g(x) are not calculated.

 The primitive polynomial f(x) is specified by its primitive polynomial.

 The lower order polynomial terms are stored in LSB's.
 ------------------------------------------------------------
*/

int poly_mult_max_degree_25( int a_x, int f_x )
{
    int i, g_x, shft_f;

    g_x = 0;
    shft_f = f_x;
    for (i=0; i <= 25; i++) {
      if ( (shft_f & 1) == 1 )
          g_x = g_x ^ ( a_x << i );
      shft_f >>= 1;
    }
    g_x = g_x & 0x1FFFFFF;     /* Keep bits 0 through 24 */
    return g_x;
}

/*
 ------------------------------------------------------------
 Polynomial Multiplication With Max Degree 25 for UMTS Top Polynomial
 Determine
            g(x) = ( f(x)b(x) )_{deg<25}
 where
            f(x) is a 1 + x^3 + x^25
            b(x) is a polynomial of degree < '25'
            g(x) is restricted to have degree < '25'

 The higher order terms of g(x) are not calculated.
```

```
 The primitive polynomial f(x) is specified in the UMTS Standard.

 The lower order polynomial terms are stored in LSB's.
 ------------------------------------------------------------
*/

int poly_mult_max_degree_UMTS_top( int a_x )
{
    int g_x;

    g_x = a_x;
    g_x = g_x ^ ( a_x << 3 );
    g_x = g_x ^ ( a_x << 25 );
    g_x = g_x & 0x1FFFFFF;    /* Keep bits 0 through 24 */
    return g_x;
}

/*
 ------------------------------------------------------------
 Polynomial Multiplication With Max Degree 25 for UMTS Bottom Polynomial
 Determine
            g(x) = ( f(x)b(x) )_{deg<25}
 where
            f(x) is a 1 + x + x^2 + x^3 + x^25
            b(x) is a polynomial of degree < '25'
            g(x) is restricted to have degree < '25'

 The higher order terms of g(x) are not calculated.

 The primitive polynomial f(x) is specified in the UMTS Standard.

 The lower order polynomial terms are stored in LSB's.
 ------------------------------------------------------------
*/

int poly_mult_max_degree_UMTS_bot( int a_x )
{
    int g_x;

    g_x = a_x;
    g_x = g_x ^ ( a_x << 1 );
    g_x = g_x ^ ( a_x << 2 );
    g_x = g_x ^ ( a_x << 3 );
    g_x = g_x ^ ( a_x << 25 );
    g_x = g_x & 0x1FFFFFF;    /* Keep bits 0 through 24 */
    return g_x;
}

/* ------------------------------------------------------------
   Bit Reverse a 25-bit Integer
   ------------------------------------------------------------ */

int bit_reverse_25( int g_x )
{
    int i, r_x;
    r_x = 0;
```

```
    for(i=0; i < 25; i++) {
      r_x <<= 1;
      r_x += (g_x>>i) & 1;
    }
    return r_x;
}

/*
 ---------------------------------------------------------------
 LFSR Generator for N=25 with Mask Polynomial
 ---------------------------------------------------------------
*/

#ifndef ARC
int LFSR_gen_25_mask( int f_x, int *a_x, int m_x )
{
    int i, lsb, rxor, new_msb;

    /* Calculate LSB using mask */
    rxor = m_x & *a_x;
    lsb = 0;
    for (i=0; i <= 24; i++) {
      lsb = lsb ^ (rxor & 1);
      rxor >>= 1;
    }

    /* Calculate NEW MSB */
    rxor = f_x & *a_x;
    new_msb = 0;
    for (i=0; i <= 24; i++) {
      new_msb = new_msb ^ (rxor & 1);
      rxor >>= 1;
    }

    /* Update state */
    *a_x = (*a_x>>1) ^ ( new_msb << 24 );

    return lsb;
}
#endif

/*
 ---------------------------------------------------------------
 LFSR Generator for N=25
 ---------------------------------------------------------------
*/

#ifndef ARC
int LFSR_gen_25( int f_x, int *a_x )
{
    int i, lsb, rxor, new_msb;

    /* Extract LSB */
    lsb = *a_x & 1;

    /* Calculate NEW MSB */
    rxor = f_x & *a_x;
```

```
    new_msb = 0;
    for (i=0; i <= 24; i++) {
      new_msb = new_msb ^ (rxor & 1);
      rxor >>= 1;
    }

    /* Update state */
    *a_x = (*a_x>>1) ^ ( new_msb << 24 );

    return lsb;
}
#endif

/*
 ------------------------------------------------------------
 Print Bitstring
 where 'n' is the number of bits, 1 < n <= 32
 ------------------------------------------------------------
*/

#ifndef ARC
#include <stdio.h>

void print_bitstring( char *mesg, int poly, int n )
{
    int i;
    for(i=n-1; i >= 0; i--)
      printf("%1d ", (poly>>i) & 1);
    printf(mesg);
    printf("\n");
}
#endif

/*
 ------------------------------------------------------------
 Reduction of x^power Modulo f(x) to a polynomial
 where
         f(x) = 1 + x^3 + x^25
 ------------------------------------------------------------
*/

#ifndef ARC
#include <math.h>

int reduce_25( int power )
{
    int index, result, stop;
    short int *list = (short int*) calloc( power+1, sizeof(short int) );

    for (index=power; index >= 0; index--)
      list[index] = 0;

    list[power] = 1;
    index = power;

    while (index >= 25) {
```

```
      if (list[index]) {
          list[index] = 0;
          list[index-22] = list[index-22]^1;
          list[index-25] = list[index-25]^1;
      }
      index -= 1;
    }
    result = 0;
    stop = ( power < 25 ) ? power+1 : 25;
    for (index=0; index < stop; index++)
      result += (list[index] << index);
    free(list);
    return result;
}
#endif

/*
 ---------------------------------------------------------------
 Given a polynomial g(x), calculate the value of 'k' in

           g(x) = x^k modulo f(x)
 ---------------------------------------------------------------
*/

#ifndef ARC
int revert_modulo_poly_reduction( int g_x, int f_x )
{
    int cnt = 0;
    int shft = g_x;
    while (shft != 1) {
      if ( (shft & 1) == 0 ) {
          do {
            shft >>= 1;
            cnt++;
          } while ( (shft & 1) == 0 );
      }
      else
          shft = shft ^ f_x;
    }
    return cnt;
}
#endif

/*
 ---------------------------------------------------------------
 Print a polynomial as a sum of powers of 'x'
 ---------------------------------------------------------------
*/

#ifndef ARC
void print_poly_25( int g_x )
{
    int i, bit;
    for(i=0; i < 25; i++) {
      bit = (g_x>>i)&1;
      if (bit) {
          if (i==0)
```

```
            printf("1");
          else
            printf(" + x^%d",i);
        }
      }
      printf("\n");
}
#endif
```

```
/*
$Id: galois.h,v 1.4 2000/03/21 14:49:36 rollins Exp $

galois.h

(c) 2000 Chameleon Systems Inc.
    Algorithms in Reconfigurable Silicon

Mark Rollins
14-Mar-2000
*/

#ifndef _galios_h_

/* ------------------------------------------------------
   ARC Routines:
   ------------------------------------------------------ */

int bit_reverse_25(int g_x);

int poly_mult_max_degree_UMTS_top( int a_x );

int poly_mult_max_degree_UMTS_bot( int a_x );

int poly_mult_modulo_fx_2p25( int a_x, int b_x, int f_x );

int poly_divide_max_degree_25( int g_x, int f_x );

int poly_mult_max_degree_25( int a_x, int f_x );


/* ------------------------------------------------------
   Solaris/Debugging Routines:
   ------------------------------------------------------ */
#ifndef ARC

int LFSR_gen_25_mask( int f_x, int *a_x, int m_x );
```

```
int LFSR_gen_25( int f_x, int *a_x );

void print_bitstring( char *mesg, int poly, int n );

int revert_modulo_poly_reduction( int g_x, int f_x );

void print_poly_25( int g_x );

int reduce_25( int power );

#endif

#define _galois_h_ 1
#endif
```

```
/*
 $Id: galois_arc.c,v 1.3 2000/03/21 00:19:27 rollins Exp $

 galois_arc.c

 (c) 2000 Chameleon Systems Inc.
     Algorithms in Reconfigurable Silicon

 Mark Rollins
 14-Mar-2000
*/

#include "galois.h"

#define N_bits 1000

#define mask  0x0040090
#define poly1 0x2000009
#define user  0x1000000

int main( int argc, char **argv )
{
    int a1x, a1x_rev, a2x, a2x_rev;
    int g1x, g2x;

    a1x_rev = user;

    /* Determine new seed required to produce a delayed
       version of the LFSR sequence
    */
    a1x = bit_reverse_25( a1x_rev );
    g1x = poly_mult_max_degree_UMTS_top( a1x );
    g2x = poly_mult_modulo_fx_2p25( g1x, mask, poly1 );
    a2x = poly_divide_max_degree_25( g2x, poly1 );
    a2x_rev = bit_reverse_25( a2x );
}
```

```
/*
 $Id: galois.h,v 1.4 2000/03/21 14:49:36 rollins Exp $

 galois.h

 (c) 2000 Chameleon Systems Inc.
     Algorithms in Reconfigurable Silicon

 Mark Rollins
 14-Mar-2000
*/

#ifndef _galios_h_

/* ------------------------------------------------------------
   ARC Routines:
   ------------------------------------------------------------ */

int bit_reverse_25(int g_x);

int poly_mult_max_degree_UMTS_top( int a_x );

int poly_mult_max_degree_UMTS_bot( int a_x );

int poly_mult_modulo_fx_2p25( int a_x, int b_x, int f_x );

int poly_divide_max_degree_25( int g_x, int f_x );

int poly_mult_max_degree_25( int a_x, int f_x );


/* ------------------------------------------------------------
   Solaris/Debugging Routines:
   ------------------------------------------------------------ */
#ifndef ARC

int LFSR_gen_25_mask( int f_x, int *a_x, int m_x );

int LFSR_gen_25( int f_x, int *a_x );

void print_bitstring( char *mesg, int poly, int n );

int revert_modulo_poly_reduction( int g_x, int f_x );

void print_poly_25( int g_x );

int reduce_25( int power );

#endif

#define _galois_h_ 1
#endif
```

```
/*
 $Id: galois_tst.c,v 1.2 2000/03/21 00:17:59 rollins Exp $

 galois_tst.c

 (c) 2000 Chameleon Systems Inc.
     Algorithms in Reconfigurable Silicon

 Mark Rollins
 20-Mar-2000
*/

#include "galois.h"

#define N_bits 1000

#define mask  0x0040090
#define poly1 0x2000009

int main( int argc, char **argv )
{
    int a1x, a1x_rev, a2x, a2x_rev;
    int g1x, g2x;
    int seed_ref, seed_del, seed_msk;
    int bits_ref[N_bits], bits_del[N_bits], bits_msk[N_bits];
    int errnum;
    int i,j;

    for (i=0; i <= 0xFFFFFF; i++) {

      a1x_rev = 0x1000000 + i;

      a1x = bit_reverse_25( a1x_rev );
      g1x = poly_mult_max_degree_UMTS_top( a1x );
      g2x = poly_mult_modulo_fx_2p25( g1x, mask, poly1 );
      a2x = poly_divide_max_degree_25( g2x, poly1 );
      a2x_rev = bit_reverse_25( a2x );

      seed_ref = a1x_rev;
      seed_del = a2x_rev;
      seed_msk = a1x_rev;

      errnum = 0;

      for (j=0; j < N_bits; j++) {
          bits_ref[j] = LFSR_gen_25( poly1, &seed_ref );
          bits_msk[j] = LFSR_gen_25_mask( poly1, &seed_msk, mask );
          bits_del[j] = LFSR_gen_25( poly1, &seed_del );
          errnum += ( bits_msk[j] ^ bits_del[j] );
      }

      printf("Undelayed Reference Bits\n");
      for (j=0; j < N_bits; j++)
          printf("%1d", bits_ref[j]);
      printf("\n");

      printf("Delayed Bits - Obtained with Mask\n");
```

```
        for (j=0; j < N_bits; j++)
            printf("%1d", bits_msk[j]);
        printf("\n");

        printf("Delayed Bits - Obtained with Seed\n");
        for (j=0; j < N_bits; j++)
            printf("%1d", bits_del[j]);
        printf("\n");

        printf("Number of errors: %d\n", errnum );
    };
}
```

COMBINED DECLARATION AND POWER OF ATTORNEY FOR DESIGN PATENT APPLICATION	Attorney's Docket No. 032001-074

As a below-named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name;

I BELIEVE I AM THE ORIGINAL, FIRST AND SOLE INVENTOR (if only one name is listed below) OR AN ORIGINAL, FIRST AND JOINT INVENTOR (if more than one name is listed below) OF THE SUBJECT MATTER WHICH IS CLAIMED AND FOR WHICH A PATENT IS SOUGHT ON THE INVENTION ENTITLED:

Gold Code Generator Design

the specification of which

(check one) ☒ is attached hereto;

☐ was filed on _____ as

Application No. _____

and was amended on _____; (if applicable)

I HAVE REVIEWED AND UNDERSTAND THE CONTENTS OF THE ABOVE-IDENTIFIED SPECIFICATION, INCLUDING THE CLAIMS, AS AMENDED BY ANY AMENDMENT REFERRED TO ABOVE;

I ACKNOWLEDGE THE DUTY TO DISCLOSE TO THE OFFICE ALL INFORMATION KNOWN TO ME TO BE MATERIAL TO PATENTABILITY AS DEFINED IN TITLE 37, CODE OF FEDERAL REGULATIONS, Sec. 1.56 (as amended effective March 16, 1992);

I do not know and do not believe the said invention was ever known or used in the United States of America before my or our invention thereof, or patented or described in any printed publication in any country before my or our invention thereof or more than one year prior to said application; that said invention was not in public use or on sale in the United States of America more than one year prior to said application; that said invention has not been patented or made the subject of an inventor's certificate issued before the date of said application in any country foreign to the United States of America on any application filed by me or my legal representatives or assigns more than six months prior to said application;

I hereby claim foreign priority benefits under Title 35, United States Code Sec. 119 and Sec. 172 of any foreign application(s) for patent or inventor's certificate as indicated below and have also identified below any foreign application for patent or inventor's certificate on this invention having a filing date before that of the application(s) on which priority is claimed:

COMBINED DECLARATION AND POWER OF ATTORNEY	Attorney's Docket No. 032001-074

COUNTRY/INTERNATIONAL	APPLICATION NUMBER	DATE OF FILING (day, month, year)	PRIORITY CLAIMED
			YES_ NO_
			YES_ NO_

I hereby appoint the following attorneys and agent(s) to prosecute said application and to transact all business in the Patent and Trademark Office connected therewith and to file, prosecute and to transact all business in connection with international applications directed to said invention:

William L. Mathis	17,337	R. Danny Huntington	27,903	Gerald F. Swiss	30,113
Robert S. Swecker	19,885	Eric H. Weisblatt	30,505	Charles F. Wieland III	33,096
Platon N. Mandros	22,124	James W. Peterson	26,057	Bruce T. Wieder	33,815
Benton S. Duffett, Jr.	22,030	Teresa Stanek Rea	30,427	Todd R. Walters	34,040
Norman H. Stepno	22,716	Robert E. Krebs	25,885	Ronni S. Jillions	31,979
Ronald L. Grudziecki	24,970	William C. Rowland	30,888	Harold R. Brown III	36,341
Frederick G. Michaud, Jr.	26,003	T. Gene Dillahunty	25,423	Allen R. Baum	36,086
Alan E. Kopecki	25,813	Patrick C. Keane	32,858	Steven M. duBois	35,023
Regis E. Slutter	26,999	B. Jefferson Boggs, Jr.	32,344	Brian P. O'Shaughnessy	32,747
Samuel C. Miller, III	27,360	William H. Benz	25,952	Kenneth B. Leffler	36,075
Robert G. Mukai	28,531	Peter K. Skiff	31,917	Fred W. Hathaway	32,236
George A. Hovanec, Jr.	28,223	Richard J. McGrath	29,195		
James A. LaBarre	28,632	Matthew L. Schneider	32,814		
E. Joseph Gess	28,510	Michael G Savage	32,596		

21839

and: Joseph P. O'Malley, Reg. No. 36,226

Address all correspondence to:

21839

Robert E. Krebs
BURNS, DOANE, SWECKER & MATHIS, L.L.P.
P.O. Box 1404
Alexandria, Virginia 22313-1404

Address all telephone calls to: Joseph P. O'Malley at (650)622-2300.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

FULL NAME OF SOLE OR FIRST INVENTOR	SIGNATURE	DATE
DANIEL J. PUGH	Daniel J Pugh	10/26/00
RESIDENCE		CITIZENSHIP
1595 Rebel Way, San Jose, CA 95118		United States
POST OFFICE ADDRESS		
1595 Rebel Way, San Jose, CA 95118		
FULL NAME OF SECOND JOINT INVENTOR, IF ANY	SIGNATURE	DATE
MARK ROLLINS		
RESIDENCE		CITIZENSHIP
23 Stonepath Crescent, Stittsville, Ontario, Canada		Canada
POST OFFICE ADDRESS		
23 Stonepath Crescent, Stittsville, Ontario, Canada		